UNIVERSIDAD DE LA RIOJA

TRABAJO FIN DE ESTUDIOS

Título

Fundamentación matemática del aprendizaje automático y profundo

Autor/es

RUBÉN MAZO TOMÁS

Director/es

JESÚS MARÍA ARANSAY AZOFRA y CÉSAR DOMÍNGUEZ PÉREZ ,

Facultad

Facultad de Ciencia y Tecnología

Titulación

Grado en Matemáticas

Departamento

MATEMÁTICAS Y COMPUTACIÓN

Curso académico

2017-18

publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE GRADO

Grado en Matemáticas

FUNDAMENTACIÓN MATEMÁTICA DEL APRENDIZAJE AUTOMÁTICO Y PROFUNDO

Realizado por:

Rubén Mazo Tomás

Tutelado por:

Jesús María Aransay Azofra

César Domínguez Pérez

Logroño, julio, 2018

Resumen

En este trabajo estudiaremos las redes neuronales artificiales con un punto de vista matemático, analizando los aspectos básicos del aprendizaje automático y profundo desde dos perspectivas diferentes:

En primer lugar, las redes neuronales artificiales son modelos matemáticos inspirados en la estructura y el comportamiento biológico de las neuronas. Veremos que estos modelos vienen definidos por un conjunto de entradas, un conjunto de pesos sinápticos, funciones de agregación y de activación, y una salida.

En segundo lugar, a partir de los ingredientes anteriores y unos datos conocidos, las redes neuronales son capaces de recalcular automáticamente los pesos que definen la función de salida. Estudiaremos algunos algoritmos de entrenamiento que definen este cálculo y que pueden verse como un proceso de minimización del error cometido por la función de salida, para lo cual se deben aplicar técnicas de análisis matemático.

Abstract

In this work we will study artificial neural networks with a mathematical point of view, analyzing the basics of machine and deep learning from two different perspectives:

First, artificial neural networks are mathematical models inspired by the structure and biological behavior of neurons. We will see that these models are defined by a set of inputs, a set of synaptic weights, aggregation and activation functions, and an output.

Second, with the above ingredients and some known data, neural networks are able to automatically recalculate the weights that define the output function. We will study some training algorithms that define this calculation and can be seen as a process of minimization of the error committed by the output function, for which techniques of mathematical analysis must be applied.

Índice general

Introducción **1**

1. Fundamentos matemáticos **5**
1.1. Separabilidad lineal 5
1.2. Álgebra de Boole y funciones lógicas 6
1.3. Descenso por gradiente 7

2. Redes neuronales artificiales **11**
2.1. El perceptrón simple 12
2.2. Limitaciones del perceptrón simple 13
2.3. El perceptrón multicapa 16
2.4. Arquitectura de las redes neuronales 18

3. Reglas de aprendizaje **21**
3.1. Aprendizaje supervisado 22
3.2. Aprendizaje del perceptrón simple 22
3.3. Regla μ-LMS 28
3.4. Otras funciones de activación 30

4. Aprendizaje de redes multicapa **33**
4.1. Algoritmo de propagación hacia atrás 33
4.1.1. Modelo de RNA con dos capas 34
4.1.2. Regla delta generalizada 35
4.1.3. Expresión recursiva de la regla 39
4.2. Problemas de velocidad y convergencia 40
4.2.1. Puntos planos y nodos saturados 41
4.2.2. La tasa de aprendizaje. Inercia 42
4.2.3. Sobreajuste y regularización 43

5. Ejemplos y aplicaciones **45**
5.1. Un ejemplo con números 45
5.2. Importancia de las capas ocultas 47
5.3. NETtalk 47
5.4. ALVINN 49

Conclusiones **51**

Introducción

Desde que en 1956, el informático John McCarthy introdujera el término en una conferencia en Dartmouth, la **inteligencia artificial** ha evolucionado enormemente, sobre todo durante las últimas décadas. Aplicaciones tan variadas como el reconocimiento de voz e imagen, vehículos autónomos, el diagnóstico de enfermedades o la detección de fraudes son ya comunes en la actualidad.

El *deep learning*, traducido al español como **aprendizaje profundo**, es una rama del *machine learning* (**aprendizaje automático**), y éste a su vez una rama de la inteligencia artificial cuyo objetivo es el desarrollo de técnicas que permitan *aprender* a las máquinas. La RAE define *aprendizaje* como «adquisición del conocimiento de algo por medio del estudio, el ejercicio o la experiencia, en especial de los conocimientos necesarios para aprender algún arte u oficio». Es decir, el aprendizaje requiere de experiencia previa, no es algo que surja de la nada.

En el contexto de la informática, esto se puede traducir en proporcionar experiencia a las máquinas: encontrar algoritmos a través de los cuales un ordenador pueda reconocer **por sí solo** patrones o características a partir de ejemplos, y modificar su comportamiento en consecuencia, en lugar de hacerlo porque ha sido programado para ello con antelación. Por ejemplo, predecir si los clientes aceptarán un producto determinado con base en las valoraciones previas de dichos clientes sobre otros productos similares.

En un intento de imitar la estructura del cerebro humano, se han desarrollado las **redes neuronales artificiales**, modelos matemáticos basados en nodos interconectados que constituyen la base del aprendizaje automático. La complejidad de una red depende del número de nodos que la componen, así como de la forma en la que se organizan (número de capas, conexiones...), que denominaremos estructura de la red. Cuanto mayor es la complejidad de una red (aprendizaje profundo), más complicados son los problemas que ésta puede resolver, pero también lo es el proceso de entrenamiento que ajusta sus parámetros para resolver esos problemas.

Algunos de los usos más habituales para las redes neuronales son la resolución de aquellos problemas para los que se dispone de un conjunto de datos previos con los que entrenar a la máquina. Por ejemplo, para problemas de clasificación de datos, reconocimiento de voz, imagen y señales, o predicciones en temas tan variopintos como pueden ser el mundo financiero y la meteorología, entre otros muchos.

Un ejemplo de problema de clasificación sencillo consiste en determinar si un punto en el plano pertenece o no a una región determinada. En la Figura 1 se muestra una situación en la que dado un punto cualquiera, queremos determinar si se encuentra en la zona A o en la zona B, algo que a simple vista resulta muy sencillo para el ojo humano. En un

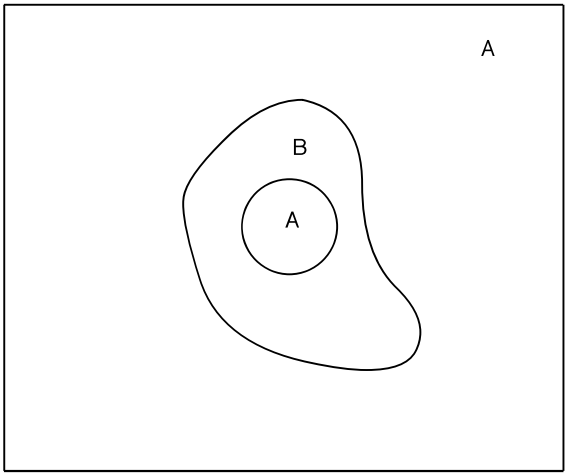

Figura 1: Problema de clasificar un punto cualquiera según su pertenencia a la zona A o a la zona B.

ordenador, sin embargo, necesitaríamos enumerar una gran cantidad de puntos con su región de pertenencia, y de todos modos la probabilidad de que un punto cualquiera estuviera en la lista sería muy pequeña (pues no es posible enumerar los infinitos puntos del plano). O en lugar de eso, deberíamos disponer de numerosas y complicadas expresiones matemáticas que le permitieran al ordenador determinar la *forma* de la zona B. Mediante el entrenamiento adecuado de una red neuronal, veremos que es posible *enseñar* al ordenador a responder si un punto está en A o en B con gran precisión, simplemente proporcionándole un conjunto finito de puntos ya clasificados para que *aprenda* de ellos.

Este ejemplo sirve también para justificar la necesidad de emplear redes neuronales de cierta complejidad si queremos resolver problemas avanzados. Veremos que no es posible entrenar a una sola neurona para resolver el problema de la Figura 1, y sin embargo sí es posible hacerlo con redes más complejas, abriendo la puerta a las redes multicapa y otros algoritmos de entrenamiento. El objetivo principal de este Trabajo es realizar un estudio de las principales técnicas de entrenamiento de redes neuronales artificiales y de los fundamentos matemáticos en los que se apoyan.

Sin entrar en demasiados detalles, las redes neuronales contienen una serie de valores conocidos como **pesos**, y es mediante la modificación de dichos pesos (a través de procesos iterativos) como se consigue entrenar a las redes neuronales para llevar a cabo con éxito tareas específicas. Bajo estos procesos aparecen técnicas y resultados con base en distintas áreas de matemáticas, como cuestiones geométricas sobre separabilidad lineal de conjuntos, algunas nociones básicas sobre funciones lógicas, y aspectos más analíticos relacionados con procesos de optimización (cálculo de mínimos) o el gradiente de funciones derivables (Capítulo 1).

Dependiendo de la complejidad de su estructura, las redes neuronales permiten resolver con éxito una mayor o menor variedad de problemas (Capítulo 2). Las redes neuronales más simples son fáciles de entrenar, pero su potencia está muy limitada. Se presentarán

diferentes algoritmos de entrenamiento y algunas de sus variantes, comenzando por redes neuronales lo más sencillas posible, de un solo nodo. En algunos casos será posible encontrar una demostración que garantice la convergencia de dicho algoritmo (Capítulo 3).

Merece especial atención el algoritmo de propagación hacia atrás, que permite de forma sencilla, aunque en general lenta, ajustar correctamente los pesos en las capas ocultas de una red neuronal compleja. No obstante, veremos que este algoritmo tiene varios inconvenientes que pueden surgir dependiendo de las características de la red y del propio problema, y mencionaremos algunas técnicas o variantes que buscan solucionarlos (Capítulo 4).

Finalmente, terminaremos con unos ejemplos prácticos del entrenamiento de redes neuronales y también describiremos brevemente algunas de las primeras aplicaciones que se llevaron a cabo con éxito (Capítulo 5).

Capítulo 1

Fundamentos matemáticos

En este capítulo se incluyen algunas definiciones y resultados conocidos de matemáticas y computación que resultarán de utilidad más adelante. La Sección 1.1 trata sobre el concepto de conjuntos linealmente separables dando una visión geométrica de su significado, y también extiende el concepto al ámbito de las funciones. La Sección 1.2 está dedicada a enunciar los aspectos básicos y notaciones del Álgebra de Boole sobre funciones lógicas que son necesarios para la comprensión de este trabajo. Para terminar, en la Sección 1.3 se introducen nociones analíticas como las derivadas parciales o el vector gradiente de funciones, gracias a los cuales se pueden desarrollar técnicas iterativas para el cálculo de mínimos. En concreto se describe el método del descenso por gradiente, que aparecerá de forma recurrente en capítulos posteriores.

1.1. Separabilidad lineal

El concepto de separabilidad lineal, tanto en funciones como en conjuntos, juega un papel importante a la hora de determinar si un problema se puede resolver con las redes neuronales más sencillas (formadas por una sola neurona). Veremos que está estrechamente relacionado con las limitaciones del llamado perceptrón simple (Sección 2.2).

Los siguientes conceptos se definen en el contexto de los espacios euclídeos, que se asumen conocidos.

Definición 1. *Sea E un espacio euclídeo de dimensión n. Se denomina* ***hiperplano*** *de E a un subespacio afín de E de dimensión $n-1$.*

Definición 2. *Sean A y B dos conjuntos de puntos en un espacio euclídeo E de dimensión n. Se dice que A y B son* ***linealmente separables*** *si existen números reales $w_1, ..., w_n, k$ tales que:*

1. $\forall a \in A$, $\sum_{i=1}^{n} w_i a_i \geq k$, *con a_i la i-ésima componente de a, y*
2. $\forall b \in B$, $\sum_{i=1}^{n} w_i b_i < k$, *con b_i la i-ésima componente de b.*

Geométricamente, A y B son linealmente separables si existe un hiperplano de E que separa los puntos de A y los de B (el hiperplano podría contener puntos de A). Dicho hiperplano viene determinado por la ecuación:

$$w_1x_1 + w_2x_2 + ... + w_nx_n = k,$$

donde los w_i y k son los números que aparecen en la definición anterior. Por ejemplo, las regiones A y B de la Figura 1 no son linealmente separables, porque no existe ninguna recta (hiperplano en $\mathbb{R}^2$) que divida el plano en dos partes dejando a un lado los puntos de A y al otro los de B.

También será de utilidad el concepto de funciones linealmente separables:

Definición 3. *Sean un espacio euclídeo E, un conjunto Y y una función:*

$$f : X \longrightarrow Y$$

donde $X \subseteq E$. Se dice que f es una ***función linealmente separable*** *si existen subconjuntos A y B de X tales que:*

1. *A y B son linealmente separables,*
2. *$A \cup B = X$, y*
3. *$f(A) \cap f(B) = \varnothing$.*

Geométricamente, una función es linealmente separable si existe un hiperplano H tal que los conjuntos de valores que toma la función a cada lado de H son disjuntos.

1.2. Álgebra de Boole y funciones lógicas

Durante el Capítulo 2 se trabajará con algunos ejemplos en los que aparecen funciones lógicas. Las herramientas necesarias para definir este tipo de funciones y describir su significado nos las proporciona el Álgebra de Boole[1]. No obstante, el estudio exhaustivo de estas herramientas (aunque es interesante) carece de sentido en el contexto de este trabajo, por lo que en esta sección nos limitaremos a enunciar los conceptos más importantes y algunas cuestiones sobre notación de funciones lógicas.

A grandes rasgos, el Álgebra de Boole se basa en la teoría de conjuntos y sirve como fundamento para el estudio de las relaciones lógicas entre variables, proporcionando un sistema simbólico compuesto por los dos siguientes tipos de elementos:

1. Variables: su característica más importante es que solo pueden tomar dos valores diferentes, que corresponden a los estados *verdadero* o *encendido* (1) y *falso* o *apagado* (0). Las denotaremos por letras $(x_1, x_2, ...)$.

2. Operaciones: son combinaciones de las variables lógicas para obtener otras variables. El Álgebra de Boole define tres operaciones básicas, a partir de las cuales se pueden obtener todas las demás operaciones:

[1] El Álgebra de Boole debe su nombre a George Boole (1947), que fue el primero en emplear técnicas algebraicas para tratar expresiones de la lógica proposicional.

x_1	x_2	$x_1 + x_2$	$x_1 x_2$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Tabla 1.1: Tabla de verdad para las operaciones lógicas suma (OR) y producto (AND).

- **Suma lógica**: relaciona dos variables de forma que su valor es 0 si y solo si ambas variables tienen valor 0, y es 1 en caso contrario. En lógica computacional se la conoce como OR y la denotaremos por el símbolo «suma»: $x_1 + x_2$.
- **Producto lógico**: relaciona dos variables de forma que su valor es 1 si y solo si ambas variables tienen valor 1, y es 0 en caso contrario. En lógica computacional se la conoce como AND y la denotaremos por el símbolo «producto»: $x_1 \cdot x_2$, o simplemente $x_1 x_2$.
- **Negación o complemento lógico**: no relaciona dos variables, sino que recibe una sola variable y toma el valor opuesto a la misma, es decir, 1 si la variable es 0, y 0 si la variable es 1. En lógica computacional se la conoce como NOT y la denotaremos por una barra sobre la variable: $\bar{x}_1$.

Las funciones lógicas surgen de la combinación de variables mediante las operaciones básicas (u otras funciones lógicas). Nótese que las tres operaciones básicas son funciones lógicas por sí mismas.

Es habitual definir las funciones lógicas mediante una **tabla de verdad**, que consiste en una presentación explícita de todas las posibles combinaciones de valores de entrada y su correspondiente valor de salida. En la Tabla 1.1 se muestran las tablas de verdad de las operaciones lógicas OR y AND. La tabla de verdad de NOT es trivial y ya ha quedado totalmente definida.

Finalmente, se dice que un conjunto de funciones lógicas es **funcionalmente completo** si a partir de ellas se puede obtener cualquier otra función lógica. En particular, las tres operaciones básicas AND, OR y NOT constituyen por definición un conjunto funcionalmente completo [18]. Veremos la importancia de este hecho en la Sección 2.3, cuando consideremos la representación de funciones lógicas mediante redes neuronales artificiales.

1.3. Descenso por gradiente

En los Capítulos 3 y 4 se describen algunos métodos para *entrenar* a las redes neuronales artificiales. Sin entrar en detalles por el momento, la mayoría de estos métodos se basan en el cálculo de mínimos en el error cometido por la red, a partir de ciertos valores que llamaremos *de entrenamiento*. El proceso que seguiremos para encontrar, o al menos tratar de encontrar dichos mínimos se conoce como **descenso por gradiente** (o descenso del gradiente, o gradiente descendente, etc.) y tiene su base en conceptos de análisis matemático, en particular algunos relacionados con el cálculo de derivadas parciales. Las nociones previas sobre topología en $\mathbb{R}^n$ y cálculo de límites se asumen conocidas.

Definición 4. *Sean una función $f : U \to \mathbb{R}^m$, con U un abierto de $\mathbb{R}^n$, y $\boldsymbol{x} = (x_1, ..., x_n) \in U$. La **derivada parcial** de f respecto a la j-ésima variable de $\boldsymbol{x}$ se define como:*

$$\frac{\partial f(\boldsymbol{x})}{\partial x_j} = \lim_{h \to 0} \frac{f(\boldsymbol{x} + h\boldsymbol{e}_j) - f(\boldsymbol{x})}{h},$$

si dicho límite existe. Aquí, $\boldsymbol{e}_j$ representa un vector de $\mathbb{R}^n$ cuyas componentes tienen todas valor 0 excepto la j-ésima, que tiene valor 1.

Definición 5. *Dados un abierto U de $\mathbb{R}^n$ y una función $f(\boldsymbol{x})$ derivable y definida en U que toma valores en $\mathbb{R}$, el **gradiente** o **vector gradiente** de f se define como el vector cuyas componentes son las derivadas parciales de f:*

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, ..., \frac{\partial f(\boldsymbol{x})}{\partial x_n} \right)^T \qquad \forall \boldsymbol{x} = (x_1, ..., x_n) \in U.$$

Geométricamente, si f toma valores en $\mathbb{R}$ podemos interpretarla como una superficie en $\mathbb{R}^{n+1}$, donde las primeras n componentes de cada punto de dicha superficie son un vector $\mathbf{x} \in U$ y la $(n+1)$-ésima componente es el valor de $f(\mathbf{x})$. Es decir, los puntos de la superficie serían de la forma:

$$\mathbf{x}^* = (x_1, ..., x_n, f(x_1, ..., x_n)), \qquad (x_1, ..., x_n) \in U \subseteq \mathbb{R}^n.$$

El gradiente de una función está asociado a la dirección en la que se producen los mayores cambios en la función. En términos de la superficie descrita anteriormente, el gradiente determina la dirección hacia la cual la *pendiente* del vector tangente a la superficie es mayor [17]. Si nos desplazamos una pequeña distancia sobre la superficie en la dirección opuesta a dicho vector (determinada por $-\nabla f$), nos encontraremos en un punto cuyo valor de f es menor. Es decir, hemos *descendido* en la superficie hacia un punto de menor valor.

Esa es la idea básica que se sigue en el método del descenso por gradiente. Comenzando en un punto cualquiera $\mathbf{x}^k$, generalmente aleatorio, se calculan nuevos puntos $\mathbf{x}^{k+1}$ de forma que:

$$\mathbf{x}^{k+1} = \mathbf{x}^k - \mu \nabla f(\mathbf{x}^k),$$

donde μ es un factor positivo que determina la magnitud del cambio. Para valores lo suficientemente pequeños de μ, $f(\mathbf{x}^*) \leq f(\mathbf{x})$. De forma general podemos definir el método como sigue:

$$\begin{cases} \mathbf{x}^1 & \text{arbitrario o aleatorio} \\ \mathbf{x}^{k+1} = \mathbf{x}^k - \mu \nabla f(\mathbf{x}^k) & k \geq 1. \end{cases}$$

Desde un punto de vista optimista, podemos suponer que:

$$f(\mathbf{x}^1) \geq f(\mathbf{x}^2) \geq f(\mathbf{x}^3) \geq ...$$

luego el método del descenso por gradiente converge hacia un mínimo local de la función f. El interés radica en poder determinar cuándo dicho mínimo es global. Por ejemplo, si la función f tiene buenas propiedades, este hecho está garantizado:

Definición 6. *Dado un conjunto convexo V de $\mathbb{R}^n$, una función $f : V \to \mathbb{R}$ se dice* ***convexa*** *si $\forall \boldsymbol{x}, \boldsymbol{y} \in V$ y $\forall t \in [0,1]$, se tiene que:*

$$f(t\boldsymbol{x} + (1-t)\boldsymbol{y}) \leq tf(\boldsymbol{x}) + (1-t)f(\boldsymbol{y}).$$

Proposición 1. *Si f es convexa y existe un $\boldsymbol{x} \in \mathbb{R}^n$ tal que $f(\boldsymbol{x})$ es un mínimo de f, entonces dicho $f(\boldsymbol{x})$ es un mínimo* ***global*** *de f.*

Demostración. Supongamos que para cierto $\mathbf{x}$, $f(\mathbf{x})$ es un mínimo de f pero existe $\mathbf{y}$ tal que $f(\mathbf{x}) > f(\mathbf{y})$, es decir, $f(\mathbf{x})$ no es un mínimo global.

Como f es convexa, se debe cumplir que:

$$f(t\mathbf{x} + (1-t)\mathbf{y}) \leq tf(\mathbf{x}) + (1-t)f(\mathbf{y}),$$

pero f tiene un mínimo en $\mathbf{x}$, por lo que existe un entorno E de $\mathbf{x}$ tal que:

$$f(\mathbf{z}) \geq f(\mathbf{x}), \qquad \forall \mathbf{z} \in E.$$

Entonces también existe $t^* \in (0,1)$ tal que:

$$f(t^*\mathbf{x} + (1-t^*)\mathbf{y}) > t^*f(\mathbf{x}) + (1-t^*)f(\mathbf{y}),$$

que es absurdo porque f es convexa. Por tanto, no puede existir dicho $\mathbf{y}$, concluyendo que f tiene un mínimo global en $\mathbf{x}$. □

El resultado anterior prueba que en determinadas funciones el descenso por gradiente converge a un mínimo global. Esto no tiene por qué ocurrir siempre. Si la función tiene un gran número de máximos y mínimos locales, resulta complicado determinar el comportamiento del descenso por gradiente. En la Sección 4.2 se presentarán, en el contexto de las redes neuronales artificiales, algunas técnicas que tratan de resolver este problema.

Capítulo 2

Redes neuronales artificiales

Las redes neuronales artificiales, o RNA, son modelos matemáticos construidos para tratar de simular el comportamiento biológico de las neuronas, imitando su estructura y forma de organización en el cerebro.

Las neuronas biológicas reciben estímulos y emiten una respuesta, que se transmite en forma de estímulo para otras neuronas. De forma similar, cada nodo o neurona artificial de una RNA recibe una o varias entradas y origina una única salida, transmitiéndola como entrada para uno o varios nodos más. El primer modelo y el más conocido es el desarrollado por Warren McCulloch y Walter Pitts en 1943 [6], que actualmente se conoce como neurona de McCulloch-Pitts o simplemente neurona artificial.

Formalmente, se pueden definir las RNA empleando conceptos de la teoría de grafos:

Definición 7. *Un* ***grafo*** *es un par ordenado $G = (V, A)$, donde V es un conjunto no vacío de vértices o nodos, y A es un conjunto de aristas que relacionan estos nodos.*

Un ***grafo dirigido*** *o* ***digrafo*** *es un grafo $G = (V, A)$ en el que las aristas son pares ordenados (a, b) con $a, b \in V$ distintos, de forma que $(a, b) \neq (b, a)$.*

Un ***grafo (o digrafo) ponderado*** *es aquel en el que las aristas llevan asociados pesos.*

La estructura de una **red neuronal artificial** puede entenderse simplemente como un digrafo ponderado en el que los vértices representan las neuronas (o nodos) y las aristas representan las conexiones entre ellas [4]. La variación de los pesos de las conexiones, entre otros elementos, es lo que permite resolver problemas concretos con una RNA. Veremos algunos ejemplos de cómo determinar dichos pesos.

En la Sección 2.1 se define un modelo para la red neuronal más sencilla, es decir, compuesta por un solo nodo. Este modelo se conoce como perceptrón simple y permite *imitar* el comportamiento de funciones lógicas, pero tiene muchas limitaciones. En la Sección 2.2 se plantean algunos ejemplos de ellas y se enuncia un resultado que relaciona el perceptrón simple con la noción de separabilidad lineal vista en el capítulo anterior. Debido a estas limitaciones, es habitual considerar modelos con más nodos organizados en capas, de forma que los nodos de una capa estén conectados solo a las capas inmediatamente anterior (la conexión se hace *desde* ella) y posterior (*hacia* ella). A una red con estas características la llamamos perceptrón multicapa, y en la Sección 2.3 se presentan algunos ejemplos y sus ventajas frente al perceptrón simple. Finalmente, en la Sección 2.4 se tratan algunos aspectos relativos a la

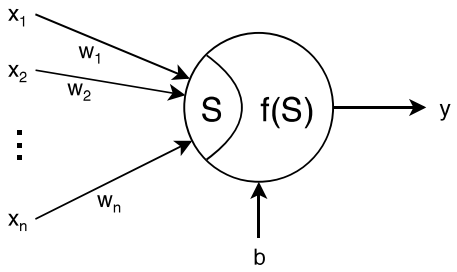

Figura 2.1: Modelo de neurona artificial.

arquitectura de las redes multicapa, sin entrar en detalles sobre el proceso de entrenamiento (esto lo veremos en los Capítulos 3 y 4).

2.1. El perceptrón simple

El modelo de neurona artificial definido a continuación es muy similar al de la neurona de McCulloch-Pitts. La Figura 2.1 es un esquema gráfico del modelo. Los elementos que lo componen son los siguientes:

1. Un vector de **entradas** $\mathbf{x} = (x_1, ..., x_n)^T$.

2. Un vector de parámetros asociados a las entradas, $\mathbf{w} = (w_1, ..., w_n)^T$ y denominados **pesos**.

3. Una **función de activación**, que determina el valor de salida según los valores de las entradas y sus pesos correspondientes: $y = f(S)$, donde el valor:

$$S = \sum_{i=1}^{n} w_i x_i = \mathbf{w}^T \mathbf{x}$$

   se denomina **función de agregación**, y en este caso es la suma ponderada de las entradas con sus pesos asociados. Nótese que con esta elección, S es una función lineal de las variables de entrada.

Un ejemplo común de elección para f es el siguiente:

$$f(S) = \begin{cases} 1 & \text{si } S \geq b \\ 0 & \text{en caso contrario.} \end{cases} \tag{2.1}$$

En este caso b recibe el nombre de *umbral* de activación, es decir, el valor mínimo que debe alcanzar S para activar la función.

Se puede considerar un conjunto de entradas que solamente tomen valores binarios, es decir $x_i \in \{0,1\}\ \forall i = 1,...,n$. El modelo obtenido de esta forma, que denominaremos **perceptrón simple**[1], se puede emplear para simular el comportamiento de algunas funciones lógicas.

Nota. En los Ejemplos 1, 2, 3 y 4 que aparecen en este capítulo, los símbolos «suma» (+) y «producto» ($\cdot$) hacen referencia a las operaciones habituales, excepto en la definición de las funciones lógicas donde se sigue la notación descrita en la Sección 1.2.

Ejemplo 1. Sea la función lógica:

$$z(x_1, x_2, x_3) = \bar{x}_1 x_2 + x_2 \bar{x}_3,$$

donde $x_1, x_2, x_3 \in \{0,1\}$, y sea la función de activación f definida en (2.1). Para determinar los valores de los pesos y el umbral, se evalúa la función de agregación $S = w_1x_1 + w_2x_2 + w_3x_3$ y se compara su valor con el de b, de modo que la función $y = f(S)$ tenga el valor deseado en cada caso. Para cada una de las 8 combinaciones de valores de entrada se obtiene una desigualdad, dando lugar al siguiente sistema:

$$\begin{cases} 0 < b \\ w_1 < b \\ w_2 \geq b \\ w_3 < b \\ w_1 + w_2 \geq b \\ w_1 + w_3 < b \\ w_2 + w_3 \geq b \\ w_1 + w_2 + w_3 < b. \end{cases}$$

Así pues, tomando, por ejemplo, como vector de entradas $\mathbf{x} = (x_1, x_2, x_3)^T$, como vector de pesos $\mathbf{w} = (-1, 2, -1)^T$ y umbral $b = \frac{1}{2}$, se obtiene la siguiente función de agregación S:

$$S = -x_1 + 2x_2 - x_3,$$

y se puede comprobar que cualquier combinación de entradas x_1, x_2 y x_3 produce la salida $f(S) = z(x_1, x_2, x_3)$. Nótese que la solución del sistema elegida es una de las infinitas posibles.

Decir que un perceptrón simple (con unos valores determinados de pesos y umbral) simula una función lógica z equivale a decir que para cualquier combinación de entradas de z, el perceptrón la clasifica correctamente, es decir, proporciona el mismo valor (0 ó 1) que z.

2.2. Limitaciones del perceptrón simple

No todas las funciones lógicas se pueden simular con un perceptrón simple. Para ilustrar este hecho, obsérvese el siguiente ejemplo.

[1] El término *perceptrón* se debe a Frank Rosenblatt (1957) [7] y puede hacer referencia a cada uno de los elementos que conforman una red neuronal artificial, o bien a la propia red neuronal.

Ejemplo 2. Sea la función lógica:

$$z(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2,$$

que corresponde a la función XOR, y toma el valor 1 si y solo si $x_1 = 0$, $x_2 = 1$ o bien $x_1 = 1$, $x_2 = 0$. Conociendo esto y evaluando $S = w_1 x_1 + w_2 x_2$ en todas las combinaciones de entradas, obtenemos el siguiente sistema de desigualdades:

$$\begin{cases} 0 < b \\ w_1 \geq b \\ w_2 \geq b \\ w_1 + w_2 < b, \end{cases}$$

que no tiene solución, ya que no existen valores mayores o iguales que b cuya suma sea menor que b. Por tanto no es posible determinar los pesos ni el umbral para que el perceptrón simple simule la función XOR.

Para un determinado número n de entradas, cada una de ellas puede tomar 2 valores distintos (0 ó 1), luego existen 2^n combinaciones diferentes de entradas. Si para cada una de ellas, una función lógica puede tomar 2 valores distintos, se tienen 2^{2^n} funciones lógicas diferentes con n entradas. A la vista del Ejemplo 2, surge la pregunta de cuáles de ellas son representables con el perceptrón simple y cuáles no. De hecho tan solo un pequeño conjunto de ellas lo son, especialmente para valores elevados de n, como veremos más adelante.

Respondiendo a la cuestión anterior, en 1969 dos científicos pioneros en inteligencia artificial, Marvin Minsky y Seymour Papert, demostraron que el perceptrón simple no se puede utilizar para resolver problemas que no sean linealmente separables [9]. Formalmente, se tiene el siguiente resultado:

Teorema 1. *Sea $z(x_1, ..., x_n)$ una función lógica. Existe un perceptrón simple que clasifica correctamente todos los posibles valores de entrada de z si y solo si z es linealmente separable.*

Demostración. Supongamos primero que existe un perceptrón simple de n entradas que simula la función z. Entonces existen valores reales $w_1, ..., w_n$ para los pesos y $b > 0$ para el umbral del perceptrón tales que:

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \geq b & \text{si } z(x_1, ..., x_n) = 1 \\ \sum_{i=1}^{n} w_i x_i < b & \text{si } z(x_1, ..., x_n) = 0. \end{cases}$$

Los valores $w_1, ..., w_n$ y b corresponden a los de la definición de conjuntos linealmente separables. Entonces, podemos definir dos conjuntos:

$$A = \{(x_1, ..., x_n) | z(x_1, ..., x_n) = 1\} \qquad y \qquad B = \{(x_1, ..., x_n) | z(x_1, ..., x_n) = 0\}$$

que cumplen:

1. A y B son linealmente separables,
2. $A \cup B = \{0, 1\}^n$, y

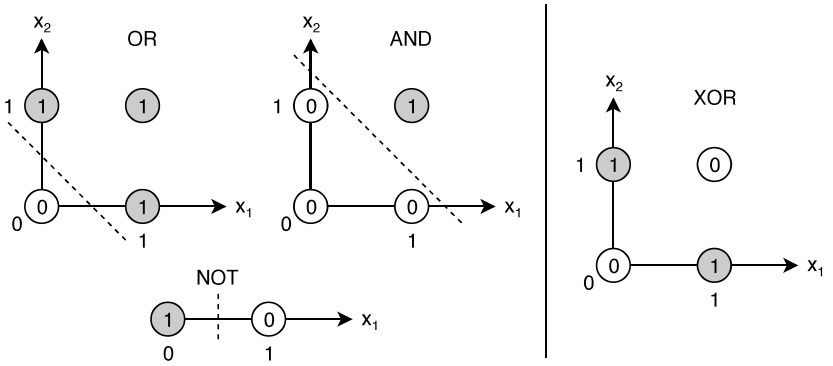

Figura 2.2: Representación gráfica de las funciones OR, AND, NOT y XOR. De las cuatro, XOR es la única que no es linealmente separable.

3. $z(A) \cap z(B) = \{1\} \cap \{0\} = \varnothing$.

Por tanto, z es una función linealmente separable.

Veamos la otra implicación. Supongamos que z es linealmente separable, y definimos A y B como en el caso anterior. Por la definición de conjuntos linealmente separables, existen $w_1, ..., w_n$ y b tales que:

$$\begin{cases} \sum_{i=1}^{n} w_i x_i \geq b & \text{si } (x_1, ..., x_n) \in A \\ \sum_{i=1}^{n} w_i x_i < b & \text{si } (x_1, ..., x_n) \in B. \end{cases}$$

Pero $w_1, ..., w_n$ y b son precisamente los pesos y el umbral de un perceptrón simple que simula correctamente la función z, luego existe dicho perceptrón simple. □

Observación. El resultado anterior no considera en ningún momento el tipo de datos de entrada de z, por lo que es generalizable a otras funciones con datos de entrada reales (que también tomen los valores 0 y 1).

Ejemplo 3. Las funciones AND, OR y NOT definidas respectivamente como:

$$z_1(x_1, x_2) = x_1 x_2 \qquad z_2(x_1, x_2) = x_1 + x_2 \qquad z_3(x_1) = \bar{x}_1$$

son linealmente separables, pero la función XOR:

$$z_4(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2$$

no lo es. Por tanto, de las cuatro, todas excepto la última son representables por un perceptrón simple. En la Figura 2.2 se puede ver la representación gráfica de los valores que toman estas funciones.

La clasificación de las funciones lógicas linealmente separables se ha llevado a cabo para valores pequeños de n, como se ve en la Tabla 2.1 donde se muestra la limitación del perceptrón simple a la hora de representar una función lógica arbitraria cuando crece el número de entradas [2].

n	Funciones lin. separables	Total de funciones (2^{2^n})	Porcentaje
1	4	4	100 %
2	14	16	87.5 %
3	104	256	40.625 %
4	1882	65536	2.872 %
5	94572	$4.3\text{x}10^{9}$	0.00002 %
6	$1.5\text{x}10^{7}$	$1.8\text{x}10^{19}$	$8.3\text{x}10^{-13}$ %
7	$8.4\text{x}10^{9}$	$3.4\text{x}10^{38}$	$2.5\text{x}10^{-29}$ %
8	$1.8\text{x}10^{13}$	$1.2\text{x}10^{77}$	$1.5\text{x}10^{-64}$ %

Tabla 2.1: Comparación del número de funciones lógicas linealmente separables de n entradas respecto al número total de funciones lógicas, para los primeros valores de n. Para $n \geq 5$ los valores son aproximados. Fuente: [2].

2.3. El perceptrón multicapa

Como se ha visto, la principal limitación del perceptrón simple es que solo permite resolver problemas linealmente separables, y los problemas reales rara vez lo son. Sin embargo, cualquier función lógica, por compleja que sea, puede representarse utilizando una combinación adecuada de funciones AND, OR y NOT, que como hemos visto, sí son linealmente separables. De modo que tiene sentido considerar una red de perceptrones simples interconectados adecuadamente, y que llamaremos **perceptrón multicapa** (por extensión del nombre), capaz de representar cualquier funcion lógica sea o no linealmente separable. La complejidad de dicha red dependerá entonces de la complejidad de la propia función en términos de las operaciones básicas (AND, OR y NOT).

En el siguiente ejemplo se muestra cómo es posible representar la función XOR, que no es linealmente separable, empleando un perceptrón de tres entradas en el que una de ellas es una función de las otras dos (concretamente AND).

Ejemplo 4. Sea la función lógica XOR:

$$z(x_1, x_2) = \bar{x}_1 x_2 + x_1 \bar{x}_2.$$

Considérese un perceptrón simple de 3 entradas x_1, x_2 y $x_1 x_2$. Los valores de los 3 pesos, $\mathbf{w} = (w_1, w_2, w_3)^T$, y b deberán ser solución del sistema:

$$\begin{cases} 0 < b \\ w_1 \geq b \\ w_2 \geq b \\ w_1 + w_2 + w_3 < b. \end{cases}$$

Nótese que el sistema tiene solo 4 desigualdades porque realmente existen solo 2 variables diferentes (x_1 y x_2), y la tercera es una función lógica de las otras. Por tanto solamente hay 4 combinaciones de valores de entrada, que corresponden a las combinaciones de valores de x_1 y x_2.

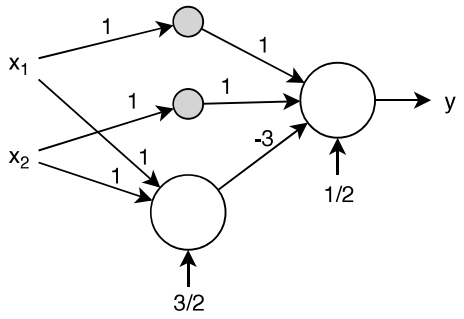

Figura 2.3: Representación de la función XOR mediante una red neuronal artificial. En los nodos sombreados no se realiza procesamiento de las entradas (función identidad).

De este modo el sistema sí tiene solución (de hecho, infinitas). Una posible solución es $\mathbf{w} = (1, 1, -3)^T$ y $b = \frac{1}{2}$, con lo que la función de activación queda de la siguiente forma:

$$f(x_1 + x_2 - 3x_1x_2) = \begin{cases} 1 & \text{si } x_1 + x_2 - 3x_1x_2 \geq \frac{1}{2} \\ 0 & \text{en caso contrario,} \end{cases}$$

donde $S = x_1 + x_2 - 3x_1x_2$ ya no es una función lineal de x_1 y x_2.

Sin embargo, considerando $x_3 = x_1x_2$, S es función lineal de sus 3 variables de entrada, aunque una depende de las otras. El valor de x_3 se obtiene a partir de la función de activación:

$$x_3 = g(x_1 + x_2) = \begin{cases} 1 & \text{si } x_1 + x_2 \geq \frac{3}{2} \\ 0 & \text{en caso contrario,} \end{cases}$$

que corresponde a la función AND con entradas x_1 y x_2.

La Figura 2.3 es la representación gráfica del Ejemplo 4. En él se muestra que formar una **red** de perceptrones simples organizados en **capas**, donde las salidas de unos constituyen las entradas de otros, permite resolver problemas que hasta ahora no tenían solución.

A continuación se incluyen una serie de definiciones con el objetivo de formalizar y aclarar algunos de los conceptos vistos hasta ahora.

Definición 8. *Un **perceptrón multicapa** es una RNA en la que los nodos están organizados en capas ordenadas, de forma que en cada capa las entradas provienen solamente de la capa anterior y las salidas se transmiten solamente a la capa siguiente.*

Definición 9. *Se dice que un perceptrón multicapa está **totalmente conectado** si cada salida de los nodos de una capa es entrada de todos los nodos de la capa siguiente. Si en alguna capa, alguna salida no es entrada de todos los nodos de la capa siguiente, se dice que el perceptrón está **localmente conectado**.*

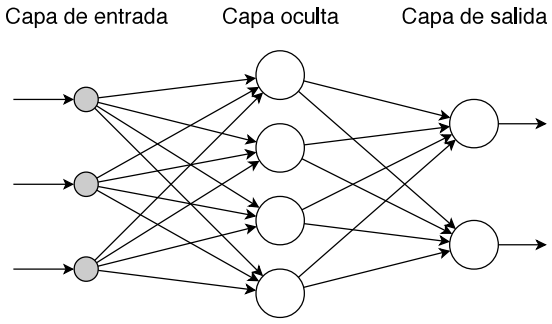

Figura 2.4: Ejemplo de perceptrón multicapa totalmente conectado. En la capa de entrada (nodos sombreados) no se realiza procesamiento de los datos.

Definición 10. *En un perceptrón multicapa con al menos 3 capas:*

1. *La **capa de entrada** está formada por todos los nodos que introducen los valores de entrada. En esta capa no se realiza ningún procesamiento de los datos.*
2. *Las **capas ocultas** son todas las capas intermedias entre la capa de entrada y la de salida. Sus entradas provienen de una capa anterior y sus salidas pasan a una capa posterior.*
3. *La **capa de salida** está formada por todos los nodos que producen los valores finales de salida.*

Por ejemplo, el modelo de la Figura 2.3 para representar la función XOR corresponde a un perceptrón multicapa localmente conectado, ya que en la capa intermedia, algunos nodos solamente reciben una de las dos entradas (x_1 o x_2). En la Figura 2.4 aparece representada la estructura de un perceptrón totalmente conectado con 3 entradas, una sola capa oculta y 2 salidas. Obsérvese que el número de nodos de las capas de entrada y salida viene determinado por el número de entradas y salidas, respectivamente, pero no hay ninguna restricción para los nodos de las capas ocultas (en el ejemplo se han escogido 4). El concepto de *capa de entrada* es meramente formal, dado que en sus nodos no se realiza procesamiento de los datos.

2.4. Arquitectura de las redes neuronales

El perceptrón multicapa es tan solo un ejemplo particular de RNA compleja. La modificación de alguna de sus características (por ejemplo, el cambio de las funciones de activación en los nodos de salida) da lugar a la creación de redes neuronales multicapa más generales. También se pueden estudiar redes con otras conexiones entre capas, conexiones dentro de una misma capa, etc., pero no profundizaremos en ellas.

Recordemos que el objetivo de diseñar este tipo de redes es poder entrenarlas después. Pero antes de iniciar cualquier proceso de entrenamiento en una red neuronal para que *aprenda*

a resolver un problema determinado, es necesario tener bien definidas las características de la red: el número de capas, el número de neuronas y la elección de la función de activación, entre otras. Estas características se conocen como **arquitectura** de la red [2].

Normalmente, la función de activación se escoge de acuerdo al tipo de valores de salida que se desean obtener. Por ejemplo, la función sigmoide para problemas de clasificación. Algunas características pueden venir determinadas por el problema, como el número de nodos en la capa de salida (que será el mismo que el número de salidas de la red). Otras, como el número de capas ocultas, quedan a elección de un diseñador, y no hay una regla general que determine cuál es el número de capas y de neuronas en cada capa que mejores resultados ofrece.

En la práctica, y conociendo el problema que se desea resolver, esta clase de cuestiones se responden mediante prueba y error: se experimenta entrenando a redes con pequeños cambios entre ellas, aumentando o reduciendo el número de capas y neuronas, y se comparan los resultados obtenidos con cada una para escoger la que mejor se ajusta al problema [2].

A primera vista, parece claro que cuantas más capas y neuronas ocultas tenga una red, más complejos serán los problemas que puede resolver, pero también se incrementa enormemente el coste computacional de los procesos de entrenamiento [15].

Capítulo 3

Reglas de aprendizaje

La característica principal de las redes neuronales artificiales, y la más interesante, es su capacidad de aprender mediante la interacción con una fuente de información. El proceso de aprendizaje se puede ver como un proceso iterativo de optimización, a través del cual se ajustan gradualmente los parámetros de la red (vectores de pesos y umbrales) para que ésta proporcione resultados lo más cercanos posible a los deseados.

Dentro del campo del aprendizaje automático (*machine learning*), se distinguen dos grandes tipos de métodos de aprendizaje: supervisado y no supervisado. El aprendizaje supervisado se trata en la Sección 3.1 junto a la noción de conjunto de entrenamiento, y es en el que nos centraremos en este trabajo. Se habla de **aprendizaje no supervisado** cuando no se conocen los valores objetivo para los datos de entrada. Se emplea con un carácter más exploratorio, por ejemplo en problemas de *clustering*, que consisten en la búsqueda de similitudes, características o patrones en los datos que los dividan en grupos claramente diferenciados. No entraremos en más detalles respecto a este tipo de aprendizaje (para ampliar información, consultar [2]).

En la Sección 3.2 se estudian algunas de las reglas de aprendizaje supervisado más habituales en un modelo similar al del perceptrón simple, pero generalizado para admitir valores de entrada reales. Veremos además que en este caso, existe un algoritmo cuya convergencia está asegurada en un número finito de pasos. Se describe también otra regla de aprendizaje basada en el método del descenso por gradiente en la Sección 3.3, y finalmente en la Sección 3.4 se generaliza esta regla para neuronas con otras funciones de activación derivables en todo su dominio.

A partir de esto, tiene sentido plantear dos direcciones en las que extender la red: agregar más nodos de salida o agregar más capas. En el primer caso, los nodos reciben datos de entrada del mismo conjunto (exactamente los mismos si la red está totalmente conectada), pero cada nodo actúa de forma independiente respecto al resto. Así pues, se puede considerar como una colección de m redes de una sola salida. La extensión a redes con varias capas (en las que aparecen capas ocultas) no es tan sencilla, y se considerará en el Capítulo 4.

3.1. Aprendizaje supervisado

En el **aprendizaje supervisado**, el objetivo es *entrenar* a la red proporcionándole un conjunto de datos ya conocidos junto a los valores de salida objetivo. De este modo la red se ajusta para asignar correctamente dichos valores de salida a nuevas entradas, cuyas salidas se desconocen. Se emplean comúnmente en problemas de clasificación (identificación de dígitos, diagnósticos, detección de fraudes...) y problemas de regresión (predicción meteorológica, de crecimiento...), dependiendo de la naturaleza de los valores de salida. En este trabajo profundizaremos en los problemas de clasificación.

El proceso de aprendizaje se conoce como **entrenamiento** de la RNA. Para poder realizarlo, es necesario disponer de valores de entrada cuyas salidas deseadas sean conocidas.

Definición 11. *Dada una RNA con n entradas y m salidas, se denomina **conjunto de entrenamiento** de la red a una colección (asumimos finita) de pares:*

$$\{(\boldsymbol{x}^1, \boldsymbol{d}^1), (\boldsymbol{x}^2, \boldsymbol{d}^2), ..., (\boldsymbol{x}^r, \boldsymbol{d}^r)\}$$

donde para cada $k = 1, 2, ..., r$ se tiene que:

1. $\boldsymbol{x}^k = (x_1^k, ..., x_n^k)^T \in \mathbb{R}^n$ *es un vector de entradas para la RNA y*
2. $\boldsymbol{d}^k = (d_1^k, ..., d_m^k)^T$ *es un vector de valores que corresponden con las salidas deseadas al introducir $\boldsymbol{x}^k$ en la RNA.*

El tipo de datos de $\mathbf{d}^k$ depende del objetivo del problema, es decir, de la salida que queramos que proporcione la red neuronal. Por ejemplo, en problemas de clasificación podemos encontrar $\mathbf{d}^k \in \{0, 1\}^m$, aunque se pueden dar otros casos (como $\mathbf{d}^k \in \mathbb{R}^m$).

A partir de un conjunto de entrenamiento, se pueden comparar las salidas deseadas con las que proporciona la red neuronal, y mediante alguna **regla de aprendizaje** ajustar progresivamente los pesos de la red para que el error cometido sea mínimo en todos los casos del conjunto de entrenamiento. La elección de una buena regla de aprendizaje tiene importancia en el proceso de entrenamiento de la red.

Conviene reservar parte del conjunto de entrenamiento, por ejemplo un 20 % de los casos. El resto de casos se emplean para entrenar a la red, y tras completar el proceso se verifica el resultado con los casos que hemos reservado y que podemos llamar *conjunto de validación* [1].

3.2. Aprendizaje del perceptrón simple

Considérese el modelo de perceptrón simple con n entradas descrito en la Sección 2.1 al que se le han realizado las siguientes modificaciones:

1. Las componentes del vector de entradas $\mathbf{x}$ pueden ser valores de $\mathbb{R}$ (no solamente 0 y 1).
2. Aunque el número de entradas sea n, consideraremos $\mathbf{x} \in \mathbb{R}^{n+1}$. La última entrada tendrá valor fijo 1 y peso asociado $-b$, siendo b el umbral en el modelo original.

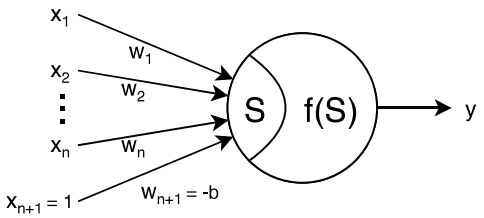

Figura 3.1: Modelo de neurona artificial donde el umbral se interpreta como una entrada más de valor 1.

Si se denotan $x_{n+1} = 1$ y $w_{n+1} = -b$, se obtiene el vector

$$\mathbf{w} = (w_1, ..., w_{n+1})^T \in \mathbb{R}^{n+1},$$

que es el vector de incógnitas del problema. Nótese que al hacer esto se está transformando el umbral b en una entrada más, por lo que formalmente, el nuevo umbral sería 0 y la función de activación sería:

$$f(S) = \begin{cases} 1 & \text{si } S \geq 0 \\ 0 & \text{en caso contrario,} \end{cases}$$

donde:

$$S = \sum_{i=1}^{n+1} w_i x_i = \sum_{i=1}^{n} w_i x_i - b.$$

Salvo el tipo de valores de entrada, el nuevo modelo es exactamente el mismo que el de la Sección 2.1 (la modificación del umbral no supone cambios en las operaciones que se realizan), de modo que seguiremos denominándolo perceptrón simple. La Figura 3.1 es el esquema gráfico del modelo a utilizar.

Sea un conjunto de entrenamiento:

$$\{(\mathbf{x}^1, d^1), (\mathbf{x}^2, d^2), ..., (\mathbf{x}^r, d^r)\}$$

donde para cada $k = 1, 2, ..., r$, $\mathbf{x}^k \in \mathbb{R}^{n+1}$ y $d^k \in \{0, 1\}$.

A través del entrenamiento, se pretende hallar los pesos $(w_1, ..., w_n)^T \in \mathbb{R}^n$ y el umbral b, tales que para todo $k = 1, 2, ..., r$ se cumpla que $y^k = d^k$, donde y^k es la salida del perceptrón y d^k es la salida deseada para el vector de entradas $\mathbf{x}^k$. Cuando ocurre esto, se dice que el perceptrón clasifica correctamente el conjunto de entrenamiento.

Calcular los pesos para que un perceptrón sea capaz de clasificar correctamente el conjunto de entrenamiento equivale a encontrar un hiperplano que divida el espacio de entrada en dos regiones, una conteniendo las entradas con $d^k = 1$ y otra conteniendo las entradas con $d^k = 0$. En otras palabras, se debe encontrar un vector $\mathbf{w}^* \in \mathbb{R}^{n+1}$ de modo que se satisfagan las siguientes relaciones:

$$\begin{cases} (\mathbf{w}^*)^T \mathbf{x}^k \geq 0 & \text{si } d^k = 1 \\ (\mathbf{w}^*)^T \mathbf{x}^k < 0 & \text{si } d^k = 0. \end{cases}$$

Así, dicho hiperplano vendrá determinado por la ecuación:

$$w_1x_1 + w_2x_2 + ... + w_nx_n = b,$$

donde los w_i, $i = 1, ..., n$ (y $w_{n+1} = -b$) son las componentes de $\mathbf{w}^*$.

Una posibilidad para hallar $\mathbf{w}^*$ es emplear el siguiente método iterativo (Rosenblatt, 1962) [7]:

$$\begin{cases} \mathbf{w}^1 \text{ elegido arbitrariamente} \\ \mathbf{w}^{j+1} = \mathbf{w}^j + \rho(d^j - y^j)\mathbf{x}^j, \quad j = 1, 2, ... \end{cases} \tag{3.1}$$

donde $\rho \in (0, 1)$ se denomina **tasa de aprendizaje** y determina la velocidad a la que *aprende* el perceptrón: cuanto más próxima a 1, mayor será la variación de los pesos en cada iteración. Si además denotamos:

$$\mathbf{z}^j = \begin{cases} \mathbf{x}^j & \text{si } d^j = 1 \\ -\mathbf{x}^j & \text{si } d^j = 0, \end{cases}$$

podemos reescribir (3.1) de la forma siguiente:

$$\begin{cases} \mathbf{w}^1 \text{ elegido arbitrariamente} & \\ \mathbf{w}^{j+1} = \mathbf{w}^j + \rho\mathbf{z}^j, & \text{si } (\mathbf{w}^j)^T\mathbf{z}^j < 0 \\ \mathbf{w}^{j+1} = \mathbf{w}^j & \text{en otro caso.} \end{cases} \tag{3.2}$$

Es decir, se trata de un método de corrección de errores: se recorre el conjunto de entrenamiento comparando los valores de salida obtenidos y los deseados, y solamente se modifican (actualizan) los pesos cuando aparece algún error. Si la salida deseada es 1 y la obtenida es 0, se aumenta el valor de los pesos según los valores de entrada, escalados por ρ. Si por el contrario, la salida deseada es 0 y la obtenida es 1, el valor de los pesos se reduce, también en función de los valores de entrada y ρ. Una vez se han comprobado todos los casos de entrenamiento, lo que podríamos denominar un *ciclo*, se comienza otra vez por el primero. El proceso se repite hasta que no aparezcan errores en ningún caso de entrenamiento.

En el análisis de cualquier regla de entrenamiento, y en particular la mostrada en (3.1), hay que considerar dos cuestiones importantes:

1. La existencia de solución (o soluciones).
2. La convergencia del algoritmo hacia la solución (en caso de que exista).

El problema de la existencia de solución ya se ha estudiado en apartados anteriores: un vector $\mathbf{w}^*$ que clasifique correctamente el conjunto de entrenamiento existe si y solo si la colección de las entradas es linealmente separable. Asumiendo esto, se tiene el siguiente resultado:

Teorema 2 (Convergencia del perceptrón)**.** *Si se tiene un conjunto de entrenamiento en el que la colección de casos de entrada es linealmente separable, entonces la regla de aprendizaje del perceptrón descrita en (3.2) converge tras un número finito de iteraciones.*

Demostración. Por la hipótesis de separabilidad lineal, existe un vector solución $\mathbf{w}^*$ que clasifica correctamente todos los casos de entrenamiento, es decir:

$$(\mathbf{w}^*)^T\mathbf{z}^k \geq 0 \qquad \forall k = 1, 2, ..., r.$$

Según la regla (3.2), solamente se realizan correcciones en $\mathbf{w}$ cuando $\mathbf{z}$ no se clasifica correctamente, es decir, cuando:

$$\mathbf{w}^T\mathbf{z} < 0.$$

Tendremos en cuenta únicamente las iteraciones donde se realiza corrección, y veremos que en cada una de ellas la diferencia entre el vector de pesos actual y la solución (o mejor dicho, el cuadrado de su norma) se va reduciendo en al menos un valor fijo. Partiendo de un vector de pesos cualquiera $\mathbf{w}^1$, y suponiendo que en una determinada iteración j del proceso la entrada $\mathbf{z}^j$ no se clasifica correctamente, podemos escribir:

$$\mathbf{w}^{j+1} - \alpha\mathbf{w}^* = \mathbf{w}^j - \alpha\mathbf{w}^* + \rho\mathbf{z}^j,$$

donde α es un valor real positivo. Entonces, tomando normas a ambos lados y desarrollando el miembro derecho:

$$\|\mathbf{w}^{j+1} - \alpha\mathbf{w}^*\|^2 = \|\mathbf{w}^j - \alpha\mathbf{w}^*\|^2 + 2\rho(\mathbf{w}^j - \alpha\mathbf{w}^*)^T\mathbf{z}^j + \rho^2\|\mathbf{z}^j\|^2.$$

Dado que $(\mathbf{w}^j)^T\mathbf{z}^j < 0$, se tiene que:

$$\|\mathbf{w}^{j+1} - \alpha\mathbf{w}^*\|^2 \leq \|\mathbf{w}^j - \alpha\mathbf{w}^*\|^2 - 2\alpha\rho(\mathbf{w}^*)^T\mathbf{z}^j + \rho^2\|\mathbf{z}^j\|^2.$$

Denotando $\beta^2 = \text{máx}_i \|\mathbf{z}^i\|^2$ y $\gamma = \text{mín}_i \rho(\mathbf{w}^*)^T\mathbf{z}^i$ (notar que $\gamma > 0$) y sustituyendo en la anterior desigualdad, se obtiene:

$$\|\mathbf{w}^{j+1} - \alpha\mathbf{w}^*\|^2 \leq \|\mathbf{w}^j - \alpha\mathbf{w}^*\|^2 - 2\alpha\gamma + \rho^2\beta^2,$$

e imponiendo que $\alpha = \beta^2/\gamma$, queda:

$$\|\mathbf{w}^{j+1} - \alpha\mathbf{w}^*\|^2 \leq \|\mathbf{w}^j - \alpha\mathbf{w}^*\|^2 - (2 - \rho^2)\beta^2.$$

Como $2 - \rho^2 > 1$, podemos simplificar aún más:

$$\|\mathbf{w}^{j+1} - \alpha\mathbf{w}^*\|^2 \leq \|\mathbf{w}^j - \alpha\mathbf{w}^*\|^2 - \beta^2,$$

que equivale a:

$$\|\mathbf{w}^j - \alpha\mathbf{w}^*\|^2 - \|\mathbf{w}^{j+1} - \alpha\mathbf{w}^*\|^2 \geq \beta^2.$$

La desigualdad anterior muestra que en cada corrección, el cuadrado de la distancia entre $\mathbf{w}^j$ y $\alpha\mathbf{w}^*$ se reduce en al menos β^2. Tras j correcciones:

$$\|\mathbf{w}^1 - \alpha\mathbf{w}^*\|^2 - \|\mathbf{w}^{j+1} - \alpha\mathbf{w}^*\|^2 \geq j\beta^2.$$

Por tanto, la corrección de errores debe terminar después de no más de M iteraciones (contando solo aquellas en las que se realiza corrección), donde:

$$M = \frac{\|\mathbf{w}^1 - \alpha\mathbf{w}^*\|^2}{\beta^2}.$$

Si además suponemos que $\mathbf{w}^1 = (0, ..., 0)^T$:

$$M = \frac{\alpha^2\|\mathbf{w}^*\|^2}{\beta^2} = \frac{\beta^2\|\mathbf{w}^*\|^2}{\gamma^2}.$$

M es un valor finito que depende de la tasa de aprendizaje ρ. Cuanto mayor sea ρ menor será M, y viceversa. □

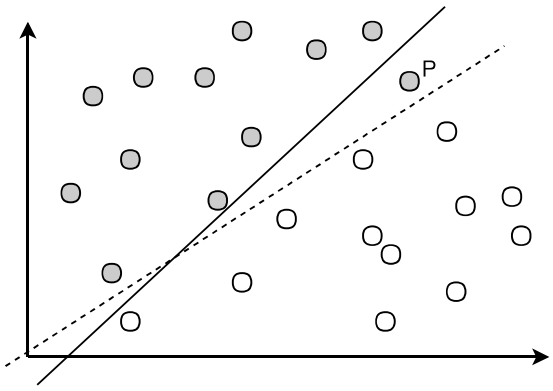

Figura 3.2: Posible solución proporcionada por un perceptrón (recta continua) para clasificar un conjunto de puntos en el plano. La adición de un nuevo punto P demuestra que existen mejores soluciones (recta discontinua).

Existen otras demostraciones equivalentes a la anterior. Algunas versiones muy similares a la aquí descrita se pueden encontrar en [2] y [4].

Algoritmo 1 (Regla del perceptrón simple)**.** El algoritmo de entrenamiento del perceptrón simple se puede resumir en los siguientes pasos:

1. Inicializar el valor de todos los pesos (por ejemplo, 0 o aleatorio).
2. Determinar la tasa de aprendizaje ρ.
3. Seleccionar un patrón de entrada $\{x_1^k, ..., x_n^k\}$ del conjunto de entrenamiento y calcular la salida proporcionada por la red con los pesos actuales.
4. Calcular el error cometido respecto a la salida deseada d^k: $d^k - y^k$.
5. Calcular el nuevo valor de los pesos siguiendo la regla definida en (3.2).
6. Repetir el proceso desde el paso 3 con otro patrón de entrada.

El proceso finaliza cuando se recorre todo el conjunto de entrenamiento sin realizar cambios en los pesos (lo que significa que todos los casos de entrenamiento son clasificados correctamente). Sin embargo, como el conjunto de entrenamiento es finito, existen infinitas soluciones (hiperplanos que separan correctamente los puntos de entrada), y el entrenamiento no garantiza que la solución obtenida sea la más adecuada para clasificar nuevos puntos fuera del conjunto de entrenamiento.

Aunque no se trate de un ejemplo real, la Figura 3.2 sirve para ilustrar este hecho. El hiperplano (recta) obtenido divide correctamente el espacio (plano) separando los puntos de las dos clases, pero puede ocurrir que al introducir un nuevo punto el perceptrón no lo clasifique correctamente. Y sin embargo existe otro hiperplano que sí lo logra, y además también clasifica correctamente el conjunto de entrenamiento.

Como en general es más probable que esto pase con puntos muy próximos a la solución proporcionada por el perceptrón, podemos tratar de *forzar* al algoritmo para que encuentre una solución más restringida, de modo que el espacio de entrada quede dividido claramente por el hiperplano, con suficiente distancia entre éste y los puntos más cercanos de cada clase.

Esto se puede conseguir modificando la regla de aprendizaje (3.2) con un valor $a > 0$ del modo siguiente:

$$\begin{cases} \mathbf{w}^1 \text{ elegido arbitrariamente} & \\ \mathbf{w}^{j+1} = \mathbf{w}^j + \rho \mathbf{z}^j, & \text{si } (\mathbf{w}^j)^T \mathbf{z}^j < a \\ \mathbf{w}^{j+1} = \mathbf{w}^j & \text{en otro caso.} \end{cases} \tag{3.3}$$

Por supuesto, siguen existiendo infinitas soluciones y esta regla tampoco garantiza que la obtenida sea la mejor, pero con ella se puede conseguir una división más apropiada del espacio de entrada de cara a clasificar nuevos puntos [2].

Otra posible regla de aprendizaje, que surge como una modificación de (3.2), es la siguiente:

$$\begin{cases} \mathbf{w}^1 \quad \text{arbitrario} & \\ \mathbf{w}^{j+1} = \mathbf{w}^j + \rho \sum_{\mathbf{z} \in Z(\mathbf{w}^j)} \mathbf{z} & j = 1, 2, ..., \end{cases} \tag{3.4}$$

donde $\rho > 0$ y $Z(\mathbf{w})$ es el conjunto de casos de entrenamiento mal clasificados empleando $\mathbf{w}$. Siguiendo esta regla, la actualización del vector de pesos se realiza teniendo en cuenta todos los casos de entrenamiento mal clasificados.

La obtención de la regla definida en (3.4) se puede explicar fácilmente considerando la función:

$$J(\mathbf{w}) = -\sum_{\mathbf{z} \in Z(\mathbf{w})} \mathbf{w}^T \mathbf{z},$$

que es positiva si existe algún caso mal clasificado, y 0 en caso contrario, ya que para todo $\mathbf{z} \in Z$, $\mathbf{w}^T \mathbf{z} < 0$.

La función J es la *función criterio* a utilizar con el método del descenso por gradiente. Intuitivamente, el proceso busca mejorar los valores de $\mathbf{w}$ *desplazándose* sobre la superficie definida por J y en la dirección con pendiente más pronunciada. En cada iteración j, los pesos se modifican de acuerdo a la llamada regla delta:

$$\mathbf{w}^{j+1} = \mathbf{w}^j + \Delta \mathbf{w}^j, \tag{3.5}$$

donde $\Delta \mathbf{w}^j$ es el valor que actualiza los pesos, proporcional al gradiente de J:

$$\Delta \mathbf{w}^j = -\rho \nabla J(\mathbf{w}^j) = -\rho \left(\frac{\partial J}{\partial w_1^j}, ..., \frac{\partial J}{\partial w_{n+1}^j} \right)^T.$$

Como para cada $i = 1, ..., n+1$ se tiene que la derivada parcial de J respecto al i-ésimo peso es:

$$\frac{\partial J}{\partial w_i} = -\frac{\partial}{\partial w_i} \left(\sum_{\mathbf{z} \in Z(\mathbf{w})} \sum_{k=1}^{n+1} w_k z_k \right) = -\sum_{\mathbf{z} \in Z(\mathbf{w})} z_i,$$

entonces el vector gradiente de J queda:

$$\nabla J(\mathbf{w}) = -\sum_{\mathbf{z}\in Z(\mathbf{w})} \mathbf{z}.$$

Y finalmente, sustituyendo los resultados anteriores en (3.5) se obtiene la regla de actualización de pesos:

$$\mathbf{w}^{j+1} = \mathbf{w}^j + \rho \sum_{\mathbf{z}\in Z(\mathbf{w}^j)} \mathbf{z},$$

que es la regla que habíamos definido en (3.4).

Mediante un razonamiento similar y empleando la función criterio:

$$J(\mathbf{w}) = -\sum_{\mathbf{z}\in Z(\mathbf{w})} (\mathbf{w}^T\mathbf{z} - a),$$

se puede deducir una nueva regla de aprendizaje a partir de (3.3). Aunque su expresión es idéntica a (3.4), no se trata de la misma regla porque en este caso los $\mathbf{z} \in Z(\mathbf{w})$ son aquellos tales que $\mathbf{w}^T\mathbf{z} < a$.

Es importante mencionar que la definición del gradiente de J en las reglas anteriores no es matemáticamente precisa. Como en la función $J(\mathbf{w})$ se producen *saltos* cada vez que $\mathbf{w}^T\mathbf{z}^k = 0$ para algún k, el gradiente no está definido en dichos puntos. Sin embargo, la naturaleza discreta de la regla (3.4) hace que la probabilidad de que algún $\mathbf{w}^j$ coincida con uno de esos puntos sea muy pequeña.

Este problema no existirá en las reglas descritas de aquí en adelante, dado que las funciones criterio empleadas para el descenso por gradiente serán funciones derivables en todo su dominio.

3.3. Regla μ-LMS

Una de las reglas de aprendizaje más analizadas y empleadas se conoce como μ-LMS (Least Mean Square). En este caso abandonaremos el problema de clasificar elementos en conjuntos linealmente separables y consideraremos un modelo de neurona artificial donde la función de salida sea la propia función de agregación[1] como se muestra en la Figura 3.3. La razón de hacer este cambio es que la nueva función de salida es diferenciable en todo su dominio, lo que no ocurre con el caso del perceptrón, y por tanto será útil para el cálculo de derivadas parciales respecto a los distintos pesos de la red. Así, la función de salida del modelo es:

$$y = f(S) = S = \mathbf{w}^T\mathbf{x} = \sum_{i=1}^{n+1} w_i x_i.$$

Sean un conjunto de entrenamiento formado por pares $(\mathbf{x}^k, d^k)$, con $k = 1, 2, ..., r$ y un vector de pesos $\mathbf{w}^p = (w_1^p, ..., w_{n+1}^p)^T$. Aquí p hace referencia a cada una de las iteraciones

[1]Este modelo fue empleado por Widrow y Hoff (1960) recibiendo el nombre de ADALINE (ADAptative LINear Element) [8].

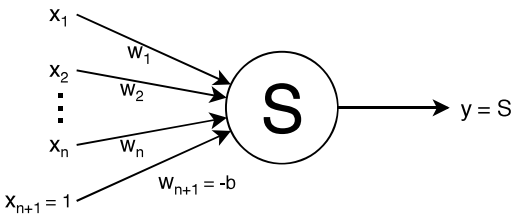

Figura 3.3: Adaptative linear element (ADALINE).

del proceso, pero lo omitiremos por simplicidad en las expresiones, denotando simplemente $\mathbf{w} = (w_1, ..., w_{n+1})^T$ en cada iteración. Con la misma consideración, se define:

$$E = \frac{1}{2}\sum_{k=1}^{r}(d^k - y^k)^2,$$

que sirve como indicador del error cometido al evaluar los r casos del conjunto de entrenamiento (aquí y^k es la salida obtenida en la iteración actual para cada caso). La función E es cuadrática en los pesos, debido a la relación lineal entre y^k y cada uno de los w_i^k con $i = 1, ..., n+1$. De hecho, su gráfica es un hiperparaboloide[2] con un único punto en el que tiene valor mínimo y que podemos denotar $\mathbf{w}^*$ [2].

Tras calcular el valor de y^k para todo el conjunto de entrenamiento, la actualización del vector de pesos se lleva a cabo siguiendo el método de la **regla delta**, basada en un descenso por gradiente sobre la función criterio E:

$$\mathbf{w}^{\text{nuevo}} = \mathbf{w} + \Delta\mathbf{w},$$

donde $\Delta\mathbf{w}$ se define en función del gradiente de E multiplicado por un factor $\mu > 0$:

$$\Delta\mathbf{w} = -\mu\nabla E = -\mu\left(\frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_{n+1}}\right)^T.$$

Nótese que para cada $i = 1, 2, ..., n+1$, las derivadas parciales de E respecto a los distintos pesos toman la forma siguiente:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\sum_{k=1}^{r}\frac{\partial(d^k - y^k)^2}{\partial y^k}\frac{\partial y^k}{\partial w_i} = -\sum_{k=1}^{r}(d^k - y^k)x_i^k.$$

Por tanto la actualización del vector de pesos y la regla general quedan:

$$\Delta\mathbf{w} = -\mu\left(-\sum_{k=1}^{r}(d^k - y^k)x_1^k, ..., -\sum_{k=1}^{r}(d^k - y^k)x_{n+1}^k\right)^T = \mu\sum_{k=1}^{r}(d^k - y^k)\mathbf{x}^k.$$

[2]Nombre generalizado de un paraboloide en n dimensiones.

$$\begin{cases} \mathbf{w}^1 & \text{arbitrario} \\ \mathbf{w}^{\text{nuevo}} = \mathbf{w} + \mu \sum_{k=1}^{r} (d^k - y^k)\mathbf{x}^k. \end{cases} \tag{3.6}$$

La regla de aprendizaje anterior se puede denominar **regla μ-LMS en paquete** (*batch*), ya que tiene en cuenta en cada iteración los r casos del conjunto de entrenamiento. Si en cada iteración solamente se trabaja con uno de los casos, se obtiene la **regla μ-LMS** incremental descrita a continuación:

$$\begin{cases} \mathbf{w}^1 & \text{arbitrario} \\ \mathbf{w}^{\text{nuevo}} = \mathbf{w} + \mu(d^k - y^k)\mathbf{x}^k, \end{cases} \tag{3.7}$$

donde en cada iteración, k hace referencia a un caso concreto del conjunto de entrenamiento.

3.4. Otras funciones de activación

Una de las principales ventajas de la regla delta con funciones de salida derivables es que se puede extender de forma natural para entrenar redes neuronales multicapa, como se verá más adelante en la Sección 4.1. Si además escogemos una función que nos permita clasificar los datos de entrada como lo hace el perceptrón simple, podremos emplear redes más complejas para resolver problemas de clasificación que no sean linealmente separables.

Considérese el mismo modelo que en la sección anterior, salvo por la elección de la función de salida. En este caso será:

$$y = f(S) = f\left(\sum_{i=1}^{n+1} w_i x_i\right)$$

donde f es una función diferenciable (nótese que $y = S$ es un caso particular de esta situación). Sea E el error cuadrático cometido en cada iteración teniendo en cuenta un solo caso k del conjunto de entrenamiento:

$$E = \frac{1}{2}(d^k - y^k)^2.$$

En este caso, las derivadas parciales de E se ven modificadas de la forma siguiente:

$$\frac{\partial E}{\partial w_i} = \frac{1}{2}\frac{\partial (d^k - y^k)^2}{\partial y^k}\frac{\partial y^k}{\partial w_i} = -(d^k - y^k)f'(S)x_i^k,$$

lo que conduce a la modificación de la regla delta siguiente:

$$\begin{cases} \mathbf{w}^1 & \text{arbitrario} \\ \mathbf{w}^{\text{nuevo}} = \mathbf{w} + \mu(d^k - f(S^k))f'(S^k)\mathbf{x}^k. \end{cases} \tag{3.8}$$

Por sus propiedades, una elección bastante habitual de f es la función sigmoide:

$$f(S) = \frac{1}{1 + e^{-S}}.$$

En la Figura 3.4 se representan las gráficas de la función sigmoide y su primera derivada. Esta elección (y la de cualquier otra función derivable con asíntotas horizontales) resulta

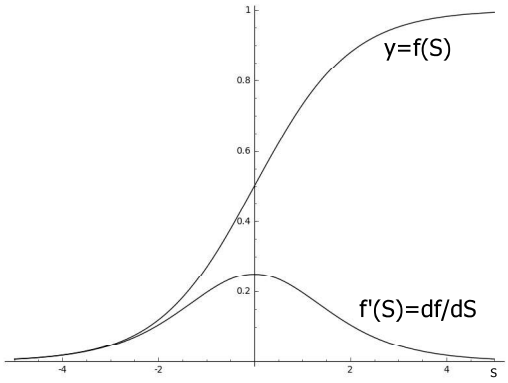

Figura 3.4: Representación gráfica de la función sigmoide y su derivada.

especialmente útil, dado que permite clasificar los datos de entrada de forma similar a como lo realiza el perceptrón simple. Si la función se aproxima a un valor a cuando $S \to -\infty$ y a otro valor b cuando $S \to +\infty$, entonces la salida de la red puede clasificar el conjunto de entradas en función de si la salida se aproxima al valor a o b. El entrenamiento de la red consistirá en encontrar un vector de pesos que produzca valores de S grandes o pequeños en cada caso. Para la función sigmoide, $a = 0$ y $b = 1$, aunque no tiene por qué ser así (por ejemplo, para la tangente hiperbólica $f(S) = \tanh S$, $a = -1$ y $b = 1$).

Por otra parte, la función sigmoide tiene la ventaja de que su derivada se puede expresar fácilmente en términos de la propia función:

$$f'(S) = \frac{e^{-S}}{(1 + e^{-S})^2} = f(S)(1 - f(S)),$$

lo que facilita su cálculo. También se pueden emplear otras funciones con propiedades similares, como $f(S) = \tanh S$, que para valores absolutos grandes de S se aproxima a los valores -1 y 1, y tiene derivada $f'(S) = 1 + f(S)^2$.

Sin embargo, una desventaja de este tipo de funciones aparece inmediatamente tras observar la Figura 3.4. En particular, nótese cómo $f'(S)$ se aproxima a 0 para valores absolutos grandes de S. Estas zonas *planas* de f hacen que la regla delta progrese muy lentamente, es decir, que en cada iteración del proceso se realicen cambios muy pequeños en el vector de pesos, incluso cuando el error cometido sea grande. Este problema se verá con mayor profundidad en la Sección 4.2.1.

Capítulo 4

Aprendizaje de redes multicapa

Hasta ahora se ha estudiado el problema del entrenamiento de redes neuronales con un solo nodo de salida, y también la posibilidad de entrenar a una red con varias salidas como si cada una fuera una red independiente. Al disponer de un conjunto de entrenamiento con valores de salida conocidos, el proceso de aprendizaje en esta clase de redes es relativamente sencillo, dado que se pueden comparar las salidas obtenidas con las deseadas y definir una *función de error* que se quiere minimizar.

No obstante, se mantiene la misma limitación que existe con el perceptrón simple: solamente es posible resolver problemas linealmente separables. Para eliminar esta limitación, se puede recurrir a RNAs más complejas añadiendo capas ocultas de nodos (como el perceptrón multicapa).

En redes con varias capas, el aprendizaje de los nodos de salida es similar a las redes de una sola capa, pero en las capas ocultas no disponemos de valores de salida deseados, por lo que el cálculo de los errores cometidos es más complicado. En la Sección 4.1 se aborda este problema generalizando la regla delta a redes con varias capas, primero para un modelo de dos capas y después tratando de obtener una expresión de la regla válida para cualquier número de capas. El algoritmo de entrenamiento basado en dicha regla se denomina comúnmente algoritmo de **propagación hacia atrás**[1] o retropropagación. El nombre hace referencia al modo en que se modifican los pesos de las capas ocultas, estimando el error cometido mediante la *propagación hacia atrás* de los errores en capas más avanzadas.

Finalmente, en la Sección 4.2 se incluyen algunos de los problemas que pueden surgir con el algoritmo de retropropagación, como la existencia de mínimos locales o regiones planas en la función de error, y se sugieren técnicas o variantes que podrían mejorarlo.

4.1. Algoritmo de propagación hacia atrás

El problema de la separabilidad lineal en el perceptrón simple ya se conocía desde finales de la década de 1970 y se habían propuesto modelos de redes neuronales combinando perceptrones como una alternativa más potente (Minsky, Papert, 1969) [9], pero no se dieron soluciones al problema de actualizar los pesos en las capas ocultas de manera efectiva.

[1] Del inglés *backpropagation*.

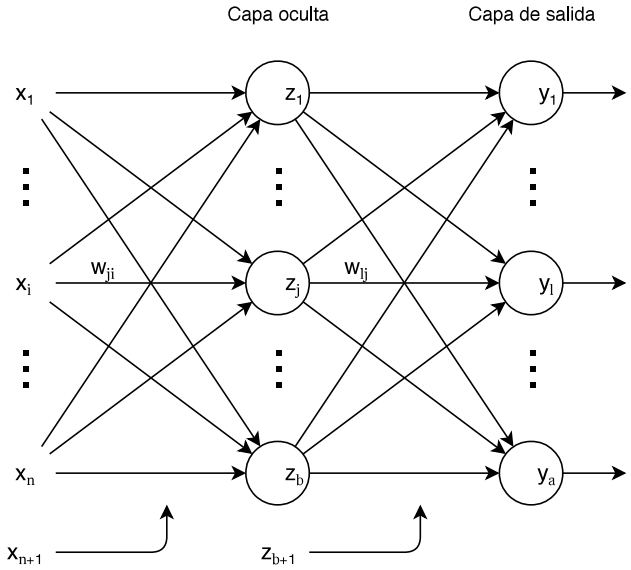

Figura 4.1: Modelo de RNA con dos capas. Consta de n entradas $\{x_1, ..., x_n\}$ y a salidas $\{y_1, ..., y_a\}$. La capa oculta tiene b nodos con salidas intermedias $\{z_1, ..., z_b\}$. Se han añadido las entradas *umbral* x_{n+1} y z_{b+1}.

En 1986, Rumelhart, Hinton y Williams [10] presentaron una forma de *trasladar* los errores medidos en las salidas de la red hacia las capas ocultas, dando lugar a una generalización de la regla delta conocida como regla de **propagación hacia atrás**. Se pueden encontrar formas levemente distintas, aunque esencialmente similares, de enunciar esta regla en [1], [2], [3], [4], [5] y [10]. La notación utilizada en ellas, incluyendo la que se emplea en este texto, puede resultar algo confusa. Por ello la Sección 4.1.1 se dedica a describir detalladamente el modelo y la notación elegidos antes de la descripción del algoritmo en la Sección 4.1.2. En la Sección 4.1.3 se presenta una solución alternativa, pero que en el fondo también es equivalente.

4.1.1. Modelo de RNA con dos capas

A continuación se describe el modelo de red neuronal que se empleará en la Sección 4.1.2. El esquema gráfico del modelo se muestra en la Figura 4.1. La RNA escogida consta de dos capas, una de salida y una oculta, con los siguientes elementos:

1. Un vector de entradas:
$$\mathbf{x} = (x_1, ..., x_n, x_{n+1})^T \in \mathbb{R}^{n+1},$$
donde la última entrada x_{n+1} actúa como umbral para los nodos de la capa oculta y tiene asignado el valor 1.

2. Un conjunto de b nodos ocultos con entrada $\mathbf{x}$. Los pesos en cada nodo se denotan como w_{ji}, con $j = 1, ..., b$ e $i = 1, ..., n + 1$ (es decir, j determina el nodo e i cada una de las entradas, de modo que w_{ji} es el i-ésimo peso de la j-ésima neurona oculta). La salida de la capa oculta es un vector:

$$\mathbf{z} = (z_1, ..., z_{b+1})^T \in \mathbb{R}^{b+1},$$

donde z_{b+1} actúa como umbral para los nodos de la capa de salida y tiene asignado el valor 1.

3. Un conjunto de a nodos de salida, con entrada $\mathbf{z}$. Los pesos en cada nodo se denotan como w_{lj}, con $l = 1, ..., a$ y $j = 1, ..., b + 1$ (es decir, l determina el nodo y j cada una de las entradas, de modo que w_{lj} es el j-ésimo peso de la l-ésima neurona de salida). La salida de la red es un vector:

$$\mathbf{y} = (y_1, ..., y_a)^T \in \mathbb{R}^a.$$

Consideraremos, por simplicidad, que la función de activación en cada capa es la misma para todos sus nodos (aunque no necesariamente debe ser así). El cálculo de las salidas *intermedias* en la capa oculta se realiza mediante una función de activación g que asumimos derivable y no lineal[2]. Una elección habitual es la función sigmoide:

$$g = \frac{1}{1 + e^{-S}},$$

donde S es la suma de las entradas por los pesos asociados, como se veía en modelos anteriores. Esta función toma valores próximos a 1 y 0 para valores de S positivos y negativos, respectivamente.

La función de activación de los nodos de la capa de salida, f, depende del objetivo del problema, es decir, del tipo de salida que se desee obtener. Si, por ejemplo, buscamos valores de salida reales, se puede tomar $f(S) = S$ en cada nodo. Otra opción bastante habitual es la elección de f de forma similar a g, si el objetivo es clasificar de algún modo los datos de entrada. Nótese que con un entrenamiento adecuado de la red, dicha clasificación sería notablemente más detallada que en redes de una sola salida, dado que en este caso se dispone de un *criterio de clasificación* diferente por cada nodo (es decir, un total de l).

4.1.2. Regla delta generalizada

Considérese el modelo de RNA descrito en la Sección 4.1.1. Sea un conjunto de entrenamiento formado por pares $(\mathbf{x}^k, \mathbf{d}^k)$, con $k = 1, ..., r$, donde los $\mathbf{x}^k = (x_1^k, ..., x_{n+1}^k)^T \in \mathbb{R}^{n+1}$ son vectores de entrada y $\mathbf{d}^k = (d_1^k, ..., d_a^k)^T$ son vectores de dimensión a (por ejemplo, $\mathbb{R}^a$ o $\{0, 1\}^a$ dependiendo del tipo de función de activación f).

Una vez más, el objetivo del entrenamiento de la red es ajustar iterativamente el valor de los pesos w, inicialmente arbitrarios o aleatorios, de modo que la salida obtenida para cada

[2]Nótese que si g fuera lineal (por ejemplo, la suma de las entradas por los pesos asociados), se podría transformar el modelo en uno de una sola capa, con lo que no habríamos eliminado la limitación de las redes monocapa.

$\mathbf{x}^k$ se aproxime lo mejor posible a $\mathbf{d}^k$. Como los valores objetivo para los nodos de salida están claramente determinados, en cada iteración del proceso definimos la función de error[3]:

$$E = \frac{1}{2}\sum_{l=1}^{a}(d_l - y_l)^2.$$

El proceso de actualización de los pesos en la capa de salida es similar al empleado en la Sección 3.3. Se denota como función de agregación para cada nodo l:

$$S_l = \sum_{j=1}^{b+1} w_{lj} z_j,$$

y como valor de los z_j (salvo $z_{b+1} = 1$) en la salida de los nodos de la capa oculta:

$$z_j = g(S_j) = g\left(\sum_{i=1}^{n+1} w_{ji} x_i\right) \qquad j = 1, 2, ..., b,$$

donde g es la función de activación en dicha capa. Entonces se tiene que:

$$\frac{\partial E}{\partial w_{lj}} = \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial S_l}\frac{\partial S_l}{\partial w_{lj}} = -(d_l - y_l) f'(S_l) z_j,$$

siendo w_{lj} cada uno de los pesos de la capa de salida (l determina el nodo y j la entrada, que corresponde a un nodo de la capa anterior). En cada iteración, al conocer los valores de salida objetivo podemos emplear directamente la regla delta para cada uno de los pesos w_{lj}, con $l = 1, ..., a$ y con $j = 1, ..., b + 1$:

$$w_{lj}^{\text{nuevo}} = w_{lj} + \Delta w_{lj} = w_{lj} - \mu\frac{\partial E}{\partial w_{lj}} = w_{lj} + \mu(d_l - y_l) f'(S_l) z_j. \tag{4.1}$$

El problema es ajustar los pesos en la capa oculta, w_{ji}, ya que la regla de aprendizaje no es tan obvia porque no se dispone de los valores de salida deseados para z_j. En lugar de eso, se trata de *propagar* los errores $d_l - y_l$ hacia atrás a través de la capa de salida, intentando estimar el valor deseado para los z_j (que variará en cada iteración). La regla de aprendizaje para las capas ocultas se puede considerar una versión generalizada de la regla delta:

$$w_{ji}^{\text{nuevo}} = w_{ji} + \Delta w_{ji} = w_{ji} - \rho\frac{\partial E}{\partial w_{ji}},$$

con ρ un valor positivo. En este caso se tiene que:

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial T_j}\frac{\partial T_j}{\partial w_{ji}} = -\frac{\partial E}{\partial z_j} g'(T_j) x_i,$$

donde T_j es la función de agregación para cada nodo j:

$$T_j = \sum_{i=1}^{n+1} w_{ji} x_i.$$

[3]Para evitar confusiones, una vez más se ha omitido el índice p que hace referencia a la iteración en curso. Naturalmente, el valor de E será diferente en cada una de ellas.

Nótese que se ha cambiado el nombre de la función de agregación para distinguir cada una de las capas. Así, en la capa de salida se denota S_l (con l indicando los nodos), y en la capa oculta se denota por T_j (con j indicando los nodos). Como E viene determinado por los errores $d_l - y_l$, se tiene que:

$$\frac{\partial E}{\partial z_j} = \frac{1}{2}\frac{\partial}{\partial z_j}\left(\sum_{l=1}^{a}(d_l - y_l)^2\right) = -\sum_{l=1}^{a}(d_l - y_l)\frac{\partial y_l}{\partial z_j} = -\sum_{l=1}^{a}(d_l - y_l)f'(S_l)w_{lj}.$$

Por tanto, los pesos en la capa oculta se actualizan de la siguiente forma:

$$w_{ji}^{\text{nuevo}} = w_{ji} + \rho\left(\sum_{l=1}^{a}(d_l - y_l)f'(S_l)w_{lj}\right)g'(T_j)x_i. \tag{4.2}$$

Comparando las reglas de actualización en (4.1) y (4.2), se puede observar que ambas son similares, pero en la segunda (correspondiente a la capa oculta) se han definido valores *dinámicos* que dependen de los pesos de la capa de salida. De esta forma, la actualización de los pesos w_{ji} en la capa oculta depende de los pesos w_{lj} en la capa de salida. Si en cada iteración denotamos:

$$d_j - z_j = \sum_{l=1}^{a}(d_l - y_l)f'(S_l)w_{lj},$$

obtenemos una expresión *aproximada* del error cometido en los nodos de la capa oculta, de forma similar a $d_l - y_l$ en la capa de salida.

Dado que los nuevos valores de w_{lj} se asumen *mejores*, utilizarlos supondría una mejora en el proceso de aprendizaje, pero entonces sería necesario volver a calcular y_l y $f'(S_l)$ para todas las salidas. Algunos autores sugieren emplear los valores iniciales en cada iteración, y no los actualizados, debido al coste operacional que conllevaría [2].

Como se veía en la Sección 2.4, dependiendo del problema a resolver puede resultar útil añadir más capas ocultas a la red. Por ello, es importante destacar que se pueden deducir reglas de actualización análogas a (4.2) en capas ocultas de mayor profundidad.

Para ilustrar este hecho considérese una modificación del modelo añadiendo una capa oculta justo al principio, antes de las entradas y la capa oculta existente (por lo que $\mathbf{x}$ ya no se considera el vector de entradas, sino que se calcula a partir de esta capa). La nueva capa tendrá vectores de entrada $V = (v_1, ..., v_{c+1})^T \in \mathbb{R}^{c+1}$ y salida $(x_1, ..., x_n)^T$, pesos w_{im} con $i = 1, ..., n$ y $m = 1, ..., c+1$ y función de activación h (diferenciable y no lineal) en todos sus nodos. La regla de actualización de los pesos w_{im} será:

$$w_{im}^{\text{nuevo}} = w_{im} + \Delta w_{im} = w_{im} - \gamma\frac{\partial E}{\partial w_{im}},$$

con γ un valor positivo. Si la función de agregación en cada nodo de esta capa es:

$$U_i = \sum_{m=1}^{c+1} w_{im}v_m,$$

entonces mediante la regla de la cadena se obtienen:

$$\frac{\partial E}{\partial w_{im}} = \frac{\partial E}{\partial x_i}\frac{\partial x_i}{\partial U_i}\frac{\partial U_i}{\partial w_{im}} = \frac{\partial E}{\partial x_i}h'(U_i)v_m,$$

$$\frac{\partial E}{\partial x_i} = \frac{1}{2}\frac{\partial}{\partial x_i}\left(\sum_{l=1}^{a}(d_l - y_l)^2\right) = -\sum_{l=1}^{a}(d_l - y_l)\frac{\partial y_l}{\partial x_i},$$

$$\frac{\partial y_l}{\partial x_i} = \frac{\partial}{\partial x_i}\left(f(S_l)\right) = f'(S_l)\frac{\partial S_l}{\partial x_i} = f'(S_l)\sum_{j=1}^{b+1} w_{lj}g'(T_j)w_{ji}.$$

Por tanto, la regla de aprendizaje en la capa oculta más profunda es:

$$w_{im}^{\text{nuevo}} = w_{im} + \gamma\left(\sum_{l=1}^{a}\sum_{j=1}^{b+1}(d_l - y_l)f'(S_l)g'(T_j)w_{lj}w_{ji}\right)h'(U_i)v_m. \tag{4.3}$$

Añadiendo más capas, el cálculo de los pesos en cada una de ellas se realizaría en función de los cálculos realizados en capas más próximas a la salida.

Algoritmo 2 (Algoritmo de propagación hacia atrás)**.** Siguiendo la notación empleada durante esta sección, el algoritmo de retropropagación para ajustar los pesos de la red se resume en los siguientes puntos:

1. Inicializar el valor de todos los pesos de la red (por ejemplo, 0 o aleatorio).
2. Determinar las tasas de aprendizaje μ, ρ... Valores pequeños ralentizarán la convergencia a la solución, pero valores grandes podrían hacer *oscilar* el algoritmo y no alcanzar nunca la solución.
3. Seleccionar un patrón de entrada $\{x_1^k, ..., x_n^k\}$ del conjunto de entrenamiento (preferiblemente aleatorio), y calcular las salidas proporcionadas por la red y_l (*propagación hacia delante*) con los pesos actuales w_{lj}.
4. Calcular el error cometido respecto a las salidas deseadas: $d_l^k - y_l^k$.
5. Calcular el nuevo valor de los pesos de la capa de salida siguiendo la regla definida en (4.1).
6. Calcular el nuevo valor de los pesos de las capa ocultas siguiendo la regla definida en (4.2) para la primera capa oculta, en (4.3) para la segunda capa oculta, etc. Como ya se ha dicho, los pesos empleados son generalmente los actuales (w_{lj},w_{ji},...) y no los nuevos obtenidos.
7. Actualizar el valor de todos los pesos de la red de acuerdo a lo obtenido en los pasos 5 y 6.
8. Comprobar la convergencia. Habitualmente se lleva a cabo evaluando una función de los errores de salida y comprobando que sea menor a un valor establecido. En caso contrario, repetir el proceso desde el paso 3.

Se trata de un proceso de aprendizaje incremental, es decir, en el que los pesos se actualizan después de la evaluación de un caso de entrenamiento. De la misma forma que con la regla μ-LMS en redes de una sola capa, se puede considerar un aprendizaje en paquete que consiste en actualizar los pesos después de haber recorrido todo el conjunto de entrenamiento.

La regla de actualización se obtiene sumando los miembros derechos en la función de error y las reglas delta, (4.1) y (4.2), para los r casos de entrenamiento:

$$E = \frac{1}{2}\sum_{k=1}^{r}\sum_{l=1}^{a}(d_l - y_l)^2,$$

$$w_{lj}^{\text{nuevo}} = w_{lj} + \mu\sum_{k=1}^{r}(d_l - y_l)f'(S_l)z_j,$$

$$w_{ji}^{\text{nuevo}} = w_{ji} + \rho\sum_{k=1}^{r}\left(\sum_{l=1}^{a}(d_l - y_l)f'(S_l)w_{lj}\right)g'(T_j)x_i.$$

4.1.3. Expresión recursiva de la regla

Las expresiones de la regla delta empleadas en la Sección 4.1.2 son bastante incómodas, por la cantidad de índices que deben tenerse en cuenta, y también a la hora de intentar obtener una expresión general para cualquier capa, aunque no es imposible. Para facilitar esta tarea, considérese un nuevo modelo de RNA con n entradas, en el que por simplicidad podemos suponer que n incluye también a la entrada *umbral*. La función de error utilizada será la misma que en los modelos anteriores:

$$E = \frac{1}{2}\sum_{l}(d_l - y_l)^2,$$

donde para cada l recorriendo todas las salidas de la red, d_l es el valor objetivo e y_l el valor real obtenido con los pesos actuales.

Podemos definir una notación para referirnos a cada una de las capas. Dada una red cualquiera, denominaremos *capa 1* a la capa de salida, *capa 2* a la primera capa oculta (entendiendo como primera la inmediatamente anterior a la salida), y así sucesivamente. Siguiendo esta notación, sea la capa J una capa cualquiera con m neuronas. Para cada $j = 1, ..., m$ y para cada $i = 1, ..., n$, los pesos w_{ji} se modifican de acuerdo al valor:

$$\Delta w_{ji} = -\mu\frac{\partial E}{\partial w_{ji}} = -\mu\frac{\partial E}{\partial S_j}\frac{\partial S_j}{\partial w_{ji}} = \mu\delta_j x_i,$$

donde $\mu > 0$, x_i es el valor proveniente de la i-ésima neurona de la capa $J+1$ y hemos definido δ_j como :

$$\delta_j = -\frac{\partial E}{\partial S_j} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial S_j} = -\frac{\partial E}{\partial y_j}f_j'(S_j).$$

Aquí f_j es la función de activación y $S_j = \sum_{i=1}^{n} w_{ji}x_i$ es la función de agregación de la j-ésima neurona de J.

El cálculo de $\frac{\partial E}{\partial y_j}$ depende del tipo de capa que sea J. Si es de salida ($J = 1$), entonces:

$$\frac{\partial E}{\partial y_j} = \frac{\partial}{\partial y_j}\left(\frac{1}{2}\sum_{j}(d_j - y_j)^2\right) = -(d_j - y_j),$$

y por tanto:

$$\delta_j = f'_j(S_j)(d_j - y_j).$$

Si por el contrario, J es una capa oculta ($J \geq 2$), entonces:

$$\frac{\partial E}{\partial y_j} = \sum_k \frac{\partial E}{\partial S_k}\frac{\partial S_k}{\partial y_j} = \sum_k (-\delta_k)\frac{\partial}{\partial y_j}\left(\sum_p w_{kp}y_p\right) = -\sum_k \delta_k w_{kj},$$

con p recorriendo las neuronas de la capa J y k recorriendo las de la capa $J-1$, que es inmediatamente posterior (para la cual las salidas de J son entradas). Nótese que aquí, S_k y δ_k hacen referencia a las neuronas de la capa $J-1$ en lugar de J. Con lo anterior, el valor de δ_j para una capa oculta es:

$$\delta_j = f'_j(S_j)\sum_k \delta_k w_{kj}.$$

Por tanto, la regla de actualización de los pesos w_{ji} de una capa cualquiera J viene determinada por $w^*_{ji} = w_{ji} + \Delta w_{ji}$, con:

$$\Delta w_{ji} = \mu\delta_j x_i, \qquad \delta_j = \begin{cases} f'_j(S_j)(d_j - y_j) & \text{en la capa de salida} \\ f'_j(S_j)\sum_k \delta_k w_{kj} & \text{en las capas ocultas.} \end{cases} \tag{4.4}$$

Los valores δ_j y δ_k son los que aportan la recursividad a la regla permitiendo expresarla de forma sencilla para cualquier capa, pero no hay que olvidar que ambos se refieren a nodos de capas distintas: δ_j corresponde a la capa J y δ_k corresponde a la capa $J-1$.

Como observación, cabe mencionar que en el caso de la capa de salida, la regla anterior es idéntica a la descrita en (4.1). Con los cambios de notación adecuados, no resultaría complicado deducir a partir de (4.4) las reglas descritas en (4.2) y (4.3) para las dos primeras capas ocultas. Igualmente, los pasos 5 y 6 del Algoritmo 2 se podrían sustituir por un paso único, en el que se calculase el nuevo valor de todos los pesos de la red siguiendo la regla (4.4).

4.2. Problemas de velocidad y convergencia

Supóngase que la red neuronal que se está entrenando tiene un total de P pesos. Independientemente de cuál sea la función de error E, podemos considerarla $E(\mathbf{w})$, es decir, una función donde $\mathbf{w} \in \mathbb{R}^P$ está formado por todos los pesos de la red. Se puede ver $E(\mathbf{w})$ como una superficie sobre la cual, en cada iteración, el algoritmo modifica el valor de $\mathbf{w}$ para que $E(\mathbf{w})$ esté cada vez más próximo a un valor mínimo. Puede ocurrir que dicho valor nunca se alcance y el algoritmo se quede oscilando sin llegar a resolver correctamente el problema, o que lo haga de forma muy lenta.

Además, si E no es una función convexa puede tener numerosos máximos, mínimos y zonas planas. Cuando el algoritmo de retropropagación converge lo hace a un mínimo *local* de la función de error utilizada [2]. En general esto es cierto para cualquier regla de aprendizaje basada en el método de descenso por gradiente, pero el problema es determinar si el valor al que converge el método es o no un mínimo *global*.

Las características y tamaño del conjunto de entrenamiento, la arquitectura de la propia red (número de nodos, capas, conexiones, funciones de activación...) o la elección de los pesos iniciales y tasas de aprendizaje son elementos que conducen a un mejor o peor entrenamiento de la red. Existen numerosos estudios enfocados en encontrar variantes que mejoren el algoritmo de retropropagación. Se ha demostrado (de manera práctica) que algunas de las técnicas que se presentan a continuación contribuyen a mejorar o acelerar la convergencia del algoritmo, e incluso a evitar que se quede *atascado* en mínimos locales de la función de error [2].

Por último, también puede ocurrir que tras el ajuste de los pesos de la red, ésta responda muy bien ante los casos de entrenamiento pero no ante casos nuevos cuyo valor (o valores) objetivo es desconocido. Esa no es la finalidad del entrenamiento y veremos que constituye un problema llamado *overfitting* que hay que tener en cuenta.

4.2.1. Puntos planos y nodos saturados

La elección del vector de pesos iniciales $\mathbf{w}^1$, o lo que es lo mismo, del punto de la superficie determinada por E desde el que comenzar el descenso por gradiente, juega un papel importante en el algoritmo de retropropagación. Por ejemplo, si $\mathbf{w}^1$ se encuentra próximo a un mínimo muy pronunciado, la convergencia será rápida hacia ese mínimo, aunque se trate de un mínimo local. En tal caso la calidad del entrenamiento dependerá de la *profundidad* de dicho mínimo respecto a la del mínimo global. Si $\mathbf{w}^1$ pertenece a una zona *plana*, sin mínimos significativos cercanos, la convergencia será muy lenta.

En la práctica, normalmente se escogen pesos iniciales de bajo valor, debido a que los pesos grandes tienden a provocar valores extremos en las funciones de activación, lo que hace a los nodos de la red insensibles al proceso de entrenamiento [10]. El uso de funciones de activación como la función sigmoide (con asíntotas horizontales para valores negativos y positivos de entrada) puede dar lugar al mismo problema.

En efecto, considérese un nodo cualquiera de una RNA con múltiples capas, con función de activación $f(S)$ sigmoide o similar. Si durante el proceso de entrenamiento, el valor absoluto de S es muy grande, la función f determinará claramente un valor próximo a una de sus asíntotas (en el caso de la sigmoide, 1 y 0). Si dicho valor difiere mucho del valor deseado (ya sea un valor conocido del conjunto de entrenamiento, para la capa de salida, o un *posible* valor correcto deducido durante el algoritmo de retropropagación, para las capas ocultas), entonces llevará un largo tiempo reajustar los pesos para que S tome un valor adecuado. Se dice que el nodo está **saturado**, y se vuelve insensible al proceso de entrenamiento.

Esto es debido a que la actualización de los pesos depende directamente de $f'(S)$, que toma valores muy próximos a 0 para valores absolutos grandes de S ralentizando la convergencia del proceso. Las correcciones en cada iteración serán pequeñas debido a que $f'(S) \approx 0$, es decir, los cambios en el vector de pesos serán casi nulos, incluso cuando el error cometido sea muy grande. Para reducir este problema se pueden *evitar* las zonas planas de f sustituyendo f' por $f' + \epsilon$, donde ϵ es una pequeña cantidad positiva. De este modo la ecuación de modificación de los pesos en (3.8) se transforma en:

$$\begin{cases} \mathbf{w}^1 \qquad \text{arbitrario} \\ \mathbf{w}^{\text{nuevo}} = \mathbf{w}^p + \mu(d^k - f(S^k))(f'(S^k) + \epsilon)\mathbf{x}^k. \end{cases} \tag{4.5}$$

4.2.2. La tasa de aprendizaje. Inercia

La velocidad de convergencia del algoritmo depende en gran medida de la elección de la tasa de aprendizaje utilizada en (4.1) y (4.2), μ y ρ respectivamente. Para evitar confusión, denominaremos μ a cualquier tasa de aprendizaje.

Intuitivamente, μ determina el impacto de las correciones sobre los pesos que se realizan tras cada iteración. Si se elige un valor de μ demasiado próximo a 0, las distintas iteraciones del proceso se aproximarán bien al *recorrido* indicado por el descenso por gradiente. Sin embargo, la modificación de los pesos, y por tanto la convergencia del algoritmo serán muy lentas, debido al gran número de iteraciones necesarias para alcanzar un mínimo local.

Por el contrario, con valores demasiado grandes de μ, el algoritmo inicialmente convergerá más deprisa hacia el valor mínimo, pero puede ocurrir que comience a oscilar sin llegar a ajustar correctamente los pesos. Lo deseable sería poder determinar μ de manera que la convergencia inicial sea rápida, pero al mismo tiempo se tenga garantía de que el proceso va a encontrar la solución en un número finito de iteraciones. Una forma de lograrlo es mediante tasas de aprendizaje adaptativas, cuyo valor se modifique de acuerdo al impacto de las correcciones en los pesos para que éstas no sean excesivamente grandes o pequeñas.

Otra posible estrategia para acelerar la convergencia del algoritmo es la adición de un término más a las reglas de aprendizaje. Este término, conocido como **inercia** (*momentum*), viene determinado por la variación que ha sufrido el vector de pesos en la iteración anterior. Si $\mathbf{w}^k = (w_1^k, ..., w_P^k)^T$ es el vector de todos los pesos de la red en la k-ésima iteración, entonces la regla de actualización considerando la inercia viene determinada por $w_i^{k+1} = w_i + \Delta w_i^k$, donde:

$$\Delta w_i^k = -\mu \frac{\partial E}{\partial w_i^k} + \alpha \Delta w_i^{k-1},$$

con α un valor generalmente entre 0 y 1.

En cierto modo, el término de inercia $\alpha \Delta w_i^{k-1}$ supone una forma de mejorar la velocidad del algoritmo en zonas casi planas de la superficie de error, y al mismo tiempo mantener tasas de aprendizaje bajas en zonas con pendientes muy pronunciadas. En efecto, aplicando recurrentemente N veces la ecuación anterior se llega a:

$$\Delta w_i^k = -\mu \frac{\partial E}{\partial w_i^k} + \alpha \Delta w_i^{k-1} = -\mu \frac{\partial E}{\partial w_i^k} - \mu\alpha \frac{\partial E}{\partial w_i^{k-1}} + \alpha \Delta w_i^{k-2} =$$

$$= ... = -\mu \sum_{n=0}^{N-1} \alpha^n \frac{\partial E}{\partial w_i^{k-n}} + \alpha^N \Delta w_i^{k-N}.$$

Si $E(\mathbf{w})$ se encuentra en una zona plana, los cambios en $\frac{\partial E}{\partial w_i}$ serán muy pequeños durante las siguientes iteraciones, por lo que si N es lo bastante grande como para despreciar la inercia, podemos aproximar:

$$\Delta w_i^k \approx -\mu \frac{\partial E}{\partial w_i^k} \sum_{n=0}^{N-1} \alpha^n \approx -\frac{\mu}{1-\alpha} \frac{\partial E}{\partial w_i^k}.$$

Por lo tanto, la inercia esencialmente está incrementando la tasa de aprendizaje efectiva en un factor $\frac{1}{1-\alpha} > 1$. Al mismo tiempo el efecto de la inercia es inapreciable en regiones donde $\frac{\partial E}{\partial w_i^k}$ toma valores muy altos.

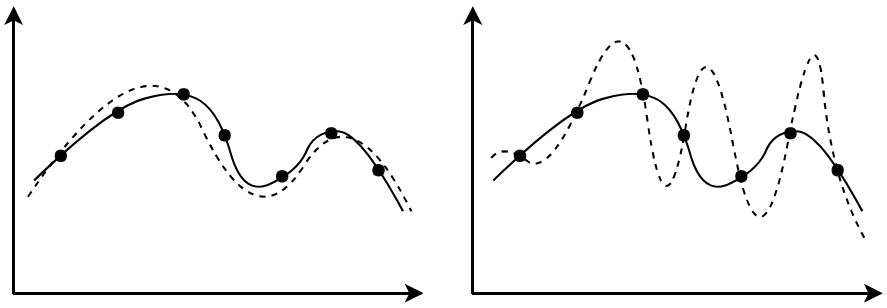

Figura 4.2: Esquema gráfico del problema del sobreajuste de los pesos. A la izquierda, red entrenada para aproximar una función. A la derecha, la red aproxima bien algunos puntos pero no el resto (sobreajuste).

4.2.3. Sobreajuste y regularización

El problema que se describe a continuación es diferente a los tratados hasta ahora, en el sentido de que ocurre cuando la convergencia del algoritmo es *demasiado buena*. Este problema, que denominaremos **sobreajuste** de los pesos de la red (*overfitting*), se visualiza más claramente en redes neuronales entrenadas para la aproximación de funciones, en lugar de la clasificación de datos de entrada. Algunos estudios sugieren que cuanto mayor es el tamaño y número de capas ocultas en la red, mayor es el riesgo de sufrir el problema del sobreajuste de pesos [1].

En la Figura 4.2 se muestra un ejemplo ficticio de en qué consiste este sobreajuste. La línea continua representa la salida deseada de la red. En la gráfica de la izquierda, la línea discontinua representa la salida real de la red para un momento concreto del entrenamiento, y en la gráfica de la derecha representa dicha salida en un momento más avanzado. Lo que ha ocurrido es que los pesos de la red se han ajustado para aproximar muy bien los datos del conjunto de entrenamiento (conjunto de puntos de la Figura 4.2), provocando errores mayores en el resto de puntos.

Una forma de evitar este hecho es detener el entrenamiento una vez que se ha alcanzado una precisión suficiente en las aproximaciones (*early-stopping*). En la Sección 3.1 se ha mencionado la conveniencia de reservar parte del conjunto de entrenamiento para realizar comprobaciones. Al no emplear algunos casos para el entrenamiento de la red, el error cometido con ellos puede ser un buen indicador de la calidad del entrenamiento. Se puede medir dicho error tras cada iteración o cada cierto número de ellas (ya que conocemos los valores objetivo), y detener el entrenamiento cuando dicho error comience a aumentar. En la Figura 4.3 se puede ver una representación gráfica de esta técnica.

Otro método menos drástico para combatir el sobreajuste se denomina **regularización** [1] y consiste en modificar la función criterio E añadiendo términos adicionales que *penalicen* a los pesos grandes. En otras palabras, si $x_1, ..., x_n$ son los datos de entrada, d e y son las

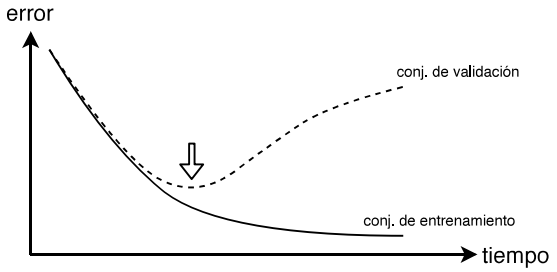

Figura 4.3: Técnica del *early-stopping*.

salidas deseada y real, y $w_1, ..., w_P$ son todos los pesos de la red, la función empleada será:

$$J(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\sum_{i=1}^{P} w_i^2,$$

donde E es la función de error original. Aquí λ representa la importancia relativa que se le da al término añadido respecto a la función de error original. Con esta modificiación, la regla de actualización de los pesos vendrá determinada, como siempre, por $w_i^{\text{nuevo}} = w_i + \Delta w_i$, con:

$$\Delta w_i = -\mu\frac{\partial J}{\partial w_i} = -\mu\frac{\partial E}{\partial w_i} - \mu\lambda w_i, \qquad \forall i = 1, ..., P.$$

El término $\mu\lambda w_i$ reduce el valor absoluto de cada peso w_i de forma proporcional a su tamaño. Esta técnica también se denomina **decaimiento de pesos** (*weight decay*) [2], y en 1987, Hinton justificó empíricamente que su utilización ayuda a evitar el sobreajuste de los pesos y mejora el proceso de entrenamiento de redes neuronales muy complejas [12].

Capítulo 5

Ejemplos y aplicaciones

Para terminar este trabajo se han incluido con fines ilustrativos algunos ejemplos prácticos y aplicaciones del uso de redes neuronales artificiales.

5.1. Un ejemplo con números

El siguiente ejemplo está inspirado en el trabajo de Matt Mazur [16]. Veamos cómo funciona paso a paso el algoritmo de propagación hacia atrás en un caso sencillo. Para simplificar los cálculos, tomaremos un conjunto de entrenamiento con un solo elemento, por ejemplo, el punto de coordenadas cartesianas $x_1 = 0{,}4$, $x_2 = 1{,}2$, con salida deseada $d = 0{,}9$, y realizaremos sobre él todas las iteraciones del proceso. En realidad esto sería lo que haríamos al emplear la regla de aprendizaje *batch* (teniendo en cuenta todos los casos de entrenamiento en cada iteración).

Consideremos el modelo de red neuronal de la Figura 5.1. El modelo consta de dos entradas, dos neuronas en la capa oculta y una sola salida. Los pesos y umbrales se han renombrado para facilitar el desarrollo de este ejemplo. Supongamos que en los tres nodos se emplea como función de activación:

$$y = f(S) = \tanh S = \frac{e^S - e^{-S}}{e^S + e^{-S}},$$

donde S es la función de agregación habitual en cada neurona, y que en toda la red tenemos una tasa de aprendizaje de $\mu = 2$. La función de error será:

$$E = \frac{1}{2}(d - y)^2.$$

En cada iteración, el cálculo de la salida de la red, y, se realiza del modo siguiente:

$$y = f(S_3) = f(w_5 z_1 + w_6 z_2 - b_3),$$

donde:

$$z_1 = f(S_1) = f(w_1 x_1 + w_2 x_2 - b_1) \quad \text{y} \quad z_2 = f(S_2) = f(w_3 x_1 + w_4 x_2 - b_2).$$

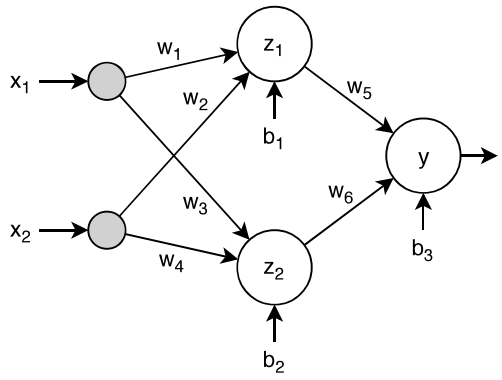

Figura 5.1: Modelo de RNA utilizado en la Sección 5.1.

La actualización de los pesos y umbrales se realiza siguiendo las reglas:

$$w_1^* = w_1 + \mu(d-y)f'(S_1)f'(S_3)w_5x_1$$

$$w_2^* = w_2 + \mu(d-y)f'(S_2)f'(S_3)w_6x_2$$

$$w_3^* = w_3 + \mu(d-y)f'(S_1)f'(S_3)w_5x_1$$

$$w_4^* = w_4 + \mu(d-y)f'(S_2)f'(S_3)w_6x_2$$

$$w_5^* = w_5 + \mu(d-y)f'(S_3)z_1$$

$$w_6^* = w_6 + \mu(d-y)f'(S_3)z_2$$

$$b_3^* = b_3 - \mu(d-y)f'(S_3)$$

$$b_1^* = b_1 - \mu(d-y)f'(S_1)f'(S_3)w_5$$

$$b_2^* = b_2 - \mu(d-y)f'(S_2)f'(S_3)w_6$$

Como pesos y umbrales iniciales se han tomado los siguientes:

$$w_1 = 1{,}2 \quad w_2 = -0{,}6 \quad w_3 = -0{,}3 \quad w_4 = 1{,}7 \quad w_5 = 0{,}5 \quad w_6 = -0{,}8$$

$$b_1 = 0{,}4 \qquad b_2 = -1{,}1 \qquad b_3 = 1{,}5$$

El proceso de entrenamiento se ha implementado utilizando SAGE. En la Tabla 5.1 se muestran los valores (aproximados) de salida obtenida y, el error cuadrático cometido E y los nuevos valores de los pesos y umbrales, para las primeras 7 iteraciones del proceso.

	w_1	w_2	w_3	w_4	w_5	w_6	b_1	b_2	b_3	y	E
0	1,2	−0,6	−0,3	1,7	0,5	−0,8	0,4	−1,1	1,5	−0,989	1,783
1	1,21	−0,6	−0,29	1,7	1,15	−0,51	0,37	−1,1	1,41	−0,988	1,782
2	1,24	−0,6	−0,26	1,7	1,16	−0,51	0,30	−1,1	1,32	−0,983	1,773
3	1,29	−0,6	−0,21	1,7	1,18	−0,50	0,19	−1,1	1,20	−0,971	1,751
4	1,37	−0,6	−0,13	1,7	1,21	−0,39	0,03	−1,1	0,99	−0,914	1,645
5	1,66	−0,61	0,16	1,7	1,29	−0,01	−0,74	−1,1	0,39	0,348	0,152
6	1,98	−0,61	0,48	1,69	2,23	0,36	−1,56	−1,1	−0,58	0,995	0,005
7	1,98	−0,61	0,48	1,69	1,98	−0,61	−1,55	−1,1	−0,58	0,947	0,001

Tabla 5.1: Entrenamiento de la red neuronal empleada en la Sección 5.1.

5.2. Importancia de las capas ocultas

Este ejemplo ha sido extraído de [2]. Recuperamos el problema que vimos en la Introducción (Figura 1) de clasificar los puntos en el plano según su pertenencia a una región A o B. Como las regiones no son linealmente separables, no es posible hacerlo mediante una red neuronal de un solo nodo. La arguitectura de la red escogida es 8-4-1, es decir, consta de tres capas, la más interna de ocho nodos, una capa intermedia de cuatro nodos y un nodo de salida. Todos ellos emplean como función de activación $f(S) = \tanh S$.

Para entrenar a la red, se ha empleado el algoritmo de retropropagación incremental con tasa de aprendizaje $\mu = 0{,}1$. El conjunto de entrenamiento se compone de 500 puntos escogidos aleatoriamente, 250 de la región A y 250 de la región B, de forma que la clasificación deseada para ellos es -1 en los puntos de A y +1 en los de B.

Tras varios cientos de ciclos sobre el conjunto de entrenamiento, se ha comprobado la convergencia del algoritmo con otros 1000 puntos aleatorios. La Figura 5.2 muestra las regiones de clasificación de cada una de las neuronas de la red tras el entrenamiento. Se ha representado un punto si y solo si la salida de la red era positiva, es decir, si la red lo clasificaba como punto de B. Las gráficas de (a) a (h) corresponden a las 8 neuronas de la capa inicial, las gráficas de (i) a (l) corresponden a las 4 neuronas intermedias, y finalmente la gráfica (m) corresponde a la salida de la red, cuyo aspecto es similar al de la Figura 1.

Se puede ver también que cada neurona adopta una *tarea* distinta, es decir, que en cada una la clasificación de un mismo punto es diferente, y es la combinación de ellas lo que permite resolver el problema. Resulta interesante observar que las neuronas de la capa más profunda, que reciben directamente las entradas de la red, funcionan como redes de un solo nodo y dividen el plano en dos regiones de forma lineal. Sin embargo, el resto de neuronas son capaces de distinguir regiones más complejas (incluso disjuntas).

5.3. NETtalk

Una de las primeras aplicaciones existosas del algoritmo de propagación hacia atrás fue desarrollada por Sejnowski y Rosenberg en 1987 [11], que lograron entrenar a una red neuronal artificial para convertir texto inglés escrito en voz. El sistema recibió el nombre de *NETtalk* y consta de dos partes: el propio modelo de red neuronal y un sintetizador de voz.

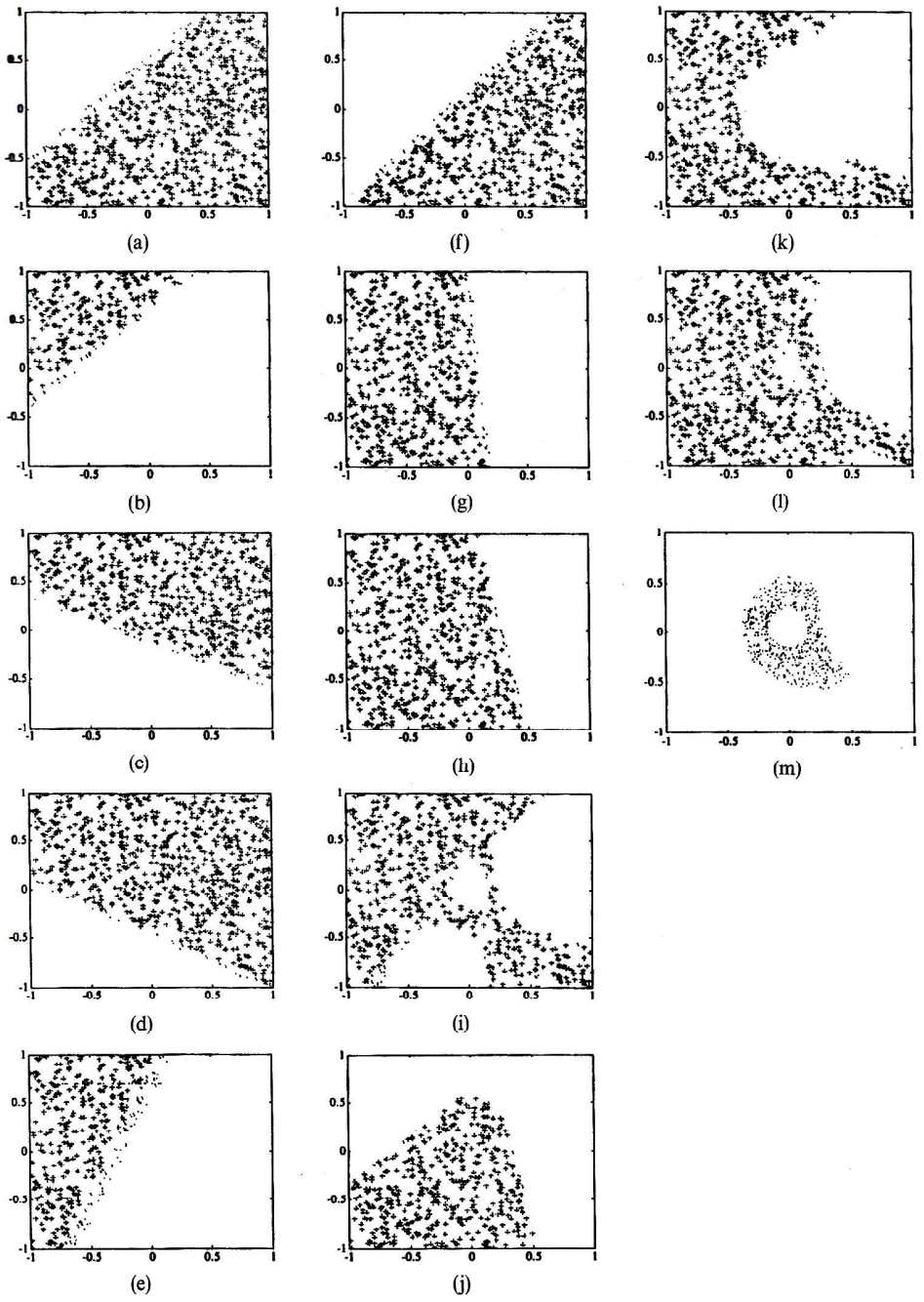

Figura 5.2: Regiones de clasificación generadas por las neuronas de la red de la Sección 5.2. (a-h) Neuronas de la primera capa oculta; (i-l) neuronas de la segunda capa oculta; (m) neurona de la capa de salida. Fuente: [2].

Figura 5.3: Vehículo de navegación autónoma NAVLAB, de la Carnegie Mellon University. Fuente: [13].

La red neuronal se compone de dos capas. La capa oculta tiene 80 neuronas que reciben un total de 203 datos divididos en 7 grupos de 29. Cada uno de estos grupos representa un carácter diferente. La capa de salida tiene 26 neuronas, cuya combinación de valores de salida sirve para dar *instrucciones* al sintetizador de voz sobre el sonido correspondiente al cuarto carácter de entrada. En toda la red se emplea como función de activación la función sigmoide.

Para el entrenamiento de la red se utilizó un texto de 1024 palabras en inglés con todos los fonemas posibles. De acuerdo a los autores, la distinción entre vocales y consonantes se producía rápidamente, aunque todas las vocales sonaban del mismo modo provocando algo parecido a un balbuceo. Más adelante la red era capaz de distinguir los límites de las palabras y tras diez ciclos sobre el conjunto de entrenamiento, el sonido producido era comprensible.

Para comprobar el resultado se comprobó el funcionamiento de la red sobre un conjunto de datos nuevo compuesto por 439 palabras, como continuación del texto empleado para el entrenamiento. La calidad del resultado se estimó en un 78 %, lo que demuestra que gran parte del *aprendizaje* a partir de una muestra simple se estaba utilizando con éxito en palabras nuevas.

5.4. ALVINN

Otro ejemplo de aplicación de las redes neuronales artificiales es ALVINN (Autonomous Land Vehicle In a Neural Network), una red entrenada para *conducir* un vehículo siguiendo una carretera mediante el empleo de una cámara de vídeo y un detector láser. En concreto, el vehículo escogido fue el NAVLAB, una ambulancia militar modificada de la Universidad de Carnegie Mellon (Figura 5.3).

La arquitectura de la red consta de una sola capa oculta con 29 neuronas y 46 en la capa de salida. La red recibe 1216 datos de entrada diferentes, distribuidos en dos cuadrículas de 30x32 y 8x32 que representan las capturas de vídeo y láser, respectivamente. De las 46 salidas, una se emplea como retroalimentación en la red para proporcionar información sobre la *intensidad* de la carretera en el momento anterior, que determina si la carretera es más clara o más oscura que el resto del terreno.

Las demás salidas determinan la dirección de giro que debe adoptar el vehículo; por ejemplo para girar a la derecha, las salidas *situadas* más a la derecha tendrán mayor peso.

Tras el entrenamiento de la red, el vehículo era capaz de recorrer por sí solo un camino de 400 metros a una velocidad de unos 2 km/h y bajo condiciones soleadas. A diferencia de otros sistemas de navegación, ALVINN era capaz de moverse en una amplia variedad de terrenos, ya que podía aprender las características más importantes de la carretera y adaptarse a ellas. Al entrenarla en diferentes carreteras, algunos nodos de la capa oculta se *especializaban* en detectar las líneas pintadas sobre el suelo, mientras que en lugares donde no había líneas se especializaban en detectar los bordes de la carretera.

Conclusiones

Son dos los motivos que me llevaron a decantarme por este tema para mi Trabajo Fin de Grado. El primero es mi interés por la inteligencia artificial, que siempre me ha resultado un tema fascinante y hasta ahora no había tenido la oportunidad de explorar con cierta profundidad.

El segundo, como estudiante de Matemáticas e Ingeniería Informática, es la relación que guarda este tema con ambas disciplinas. Al estudiar las redes neuronales artificiales desde un punto de vista matemático, he podido comprobar de primera mano algo que todos sabemos: que las matemáticas se esconden detrás de cualquier aspecto de la ciencia computacional, pero a veces no llegamos a darnos cuenta de ello. De hecho, antes de comenzar este trabajo yo tampoco era capaz de verlo, y solo hicieron falta unos minutos leyendo sobre el tema para hacerme cambiar de opinión.

Respecto al contenido del trabajo en sí mismo, hemos hecho un recorrido a través de las principales técnicas de aprendizaje automático supervisado. Hemos estudiado distintos modelos de redes neuronales artificiales, desde los más sencillos hasta modelos con varias capas, y hemos definido para cada uno de ellos algunas de las reglas de aprendizaje más comunes con objeto de proporcionar una visión general, pero solamente estamos rascando la superficie. El algoritmo de retropropagación es el primero y el más sencillo para entrenar redes multicapa, y tan solo las posibles variantes darían para otro trabajo. También existen numerosas técnicas fuera del aprendizaje supervisado (aprendizaje no supervisado, aprendizaje reforzado...) que se han quedado en el tintero.

En definitiva, realizar este trabajo me ha servido para darme cuenta de lo mucho que se ha profundizado en el ámbito del aprendizaje automático, pero también de lo mucho que todavía queda por investigar. Podríamos decir hoy en día lo mismo que dijeron Rumelhart, Hinton y Williams en 1986 [10]:

«*Finally, we should say that this work is not yet in a finished form.* [...] *However, the results to date are encouraging and we are continuing our work.*»

(«Finalmente, deberíamos decir que este trabajo todavía no tiene su forma final. [...] Sin embargo, los resultados hasta la fecha son alentadores y continuaremos trabajando.»)

Bibliografía

[1] Nikhil Buduma, *Fundamentals of Deep Learning*, O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472 (2015).

[2] Mohamad H. Hassoun, *Fundamentals of Artificial Neural Networks*, Cambridge, Massachusetts: The MIT Press (1995).

[3] D. Michie, D.J. Spiegelhalter, C.C. Taylor *Machine Learning, Neural and Statistical Classification*, Capítulo 6 (1994).

[4] Antonio Moreno y otros, *Aprendizaje Automático*, Edicions de la Universitat Politècnica de Catalunya, Capítulo 5 (1994).

[5] James A. Freeman y David M. Skapura, *Neural Networks: Algorithms, Applications and Programming Techniques*, Addison-Wesley Publishing Company (1991).

[6] Warren S. McCulloch y Walter H. Pitts, *A Logical Calculus of the Ideas Immanent in Nervous Activity*, Bulletin of Mathematical Biophysics, Vol. 5 (1943).

[7] Frank Rosenblatt, *The Perceptron: A Perceiving and Recognizing Automaton*, Cornell Aeronautical Laboratory Inc., Buffalo, N.Y. (1957).

[8] Bernard Widrow, Marcian E. Hoff, *Adaptive Switching Circuits*, Stanford Electronics Laboratories, Stanford, California (1960).

[9] Marvin L. Minsky y Seymour A. Papert, *Perceptrons: an Introduction to Computational Geometry*, Massachusetts Institute of Technology (1969), edición extendida de 1988.

[10] D. E. Rumelhart, G. E. Hinton, R. J. Williams, *Learning internal representations by error propagation*, artículo incluido en *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volumen 1, MIT Press, Cambridge, Massachusetts (1986).

[11] Terrence J. Sejnowski, Charles R. Rosenberg, *Parallel Networks that learn to Pronounce English Text*, Complex Systems Publications, Inc. (1987).

[12] G. E. Hinton, *Learning translation invariant recognition in a massively parallel network*, G. Goos y J. Hartmanis, eds., Springer-Verlag, Berlín (1987).

[13] Dean A. Pomerleau, *ALVINN: An Autonomous Land Vehicle In a Neural Network*, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA (1989).

[14] Blog de Fernando Sancho Caparrini, profesor en la Universidad de Sevilla, disponible en: http://www.cs.us.es/ fsancho/

[15] Rubén López, *Qué es y cómo funciona el Deep Learning* (2014), disponible en: https://rubenlopezg.wordpress.com/2014/05/07/que-es-y-como-funciona-deep-learning/

[16] Matt Mazur, *A Step by Step Backpropagation Example*, disponible en: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

[17] Manuel Bello Hernández, *Cálculo diferencial en varias variables*, Universidad de La Rioja (curso 2015-2016).

[18] José Luis F. Vindel, Ángeles M. Riesco, Francisco Javier D. Vegas, *Lógica computacional*, Dpto. Inteligencia Artificial, E.T.S.I. Informática, UNED (2003).